Population-Based Metaheuristics

CSI 4106 Introduction to Artificial Intelligence

Marcel Turcotte
Version: Nov 18, 2025 15:55

Preamble

Quote of the Day (1/2)

AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms, 2025-05-14.

In the media, artificial intelligence is often synonymous with deep learning, particularly concerning large language models. However, AI encompasses much more than just these technologies. Today, we are addressing the topic of genetic algorithms, which continue to be relevant in current research.

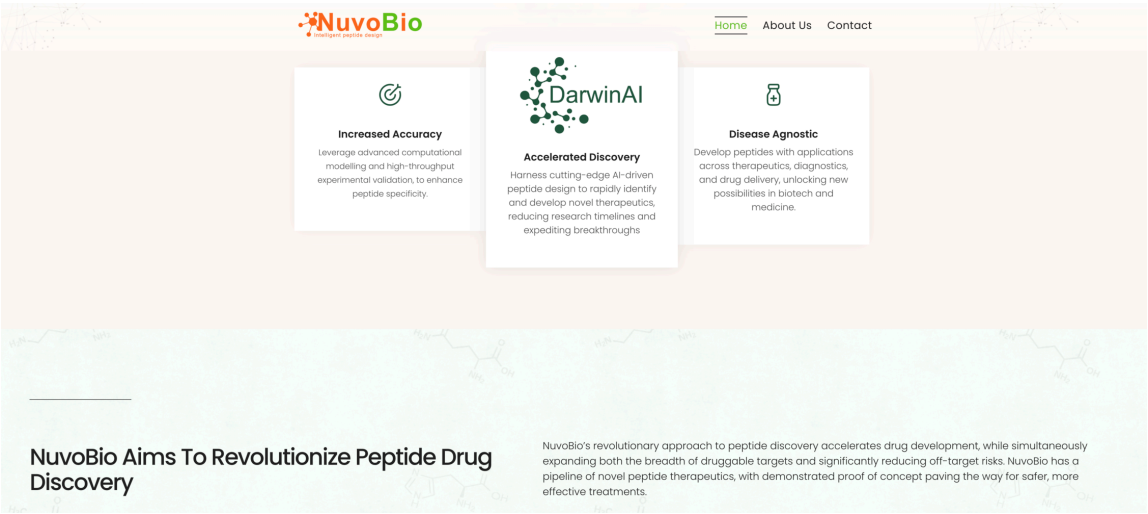
AlphaEvolve is an agent capable of evolving complete codebases and developing complex algorithms. As illustrated in the following publication, AlphaEvolve is also able to discover innovative mathematical structures.

- Mathematical exploration and discovery at scale, Georgiev et al. (2025).

- It discovered a way to multiply two 4×4 complex matrices using only 48 scalar multiplications, the first recorded improvement on this problem in 56 years.
- By devising more efficient decompositions of large matrix-multiplication subproblems, it accelerated a key Gemini training kernel by 23%, yielding a roughly 1% reduction in overall training time.
- It improved the FlashAttention kernel used in Transformer models, achieving speedups of up to 32.5%.

The algorithms presented today aim to find good solutions to optimization problems, where each solution is typically represented as a vector. However, these approaches can also be applied to learning tasks, for example to induce regular expressions or even full programs. In such cases, the solutions are no longer vectors, but rather semantic parse trees Belmadani and Turcotte (2016).

Quote of the Day (2/2)



NuvoBio: designing peptides and accelerating the development of new therapeutic treatments.

NuvoBio, an Ottawa-based company, uses AI, specifically a genetic algorithm named Darwin, to design peptides and accelerate the development of new therapeutic treatments.

The development of the Darwin tool was carried out by François Charih as part of his doctoral thesis at Carleton University, which he defended on September 12. François also holds a B.Sc. (Honours) in Biochemistry and a B.Sc.A. in Chemical Engineering with a specialization in Biotechnology, both from the University of Ottawa.

Learning Objectives

- **Understand** the definition and purpose of metaheuristics in optimization problems.
- **Learn** the principles and components of genetic algorithms (GAs).
- **Comprehend** the implementation details of GAs, including encoding, selection, crossover, mutation, and fitness evaluation.
- **Apply** GAs to solve the 0/1 knapsack problem with practical Python examples.
- **Recognize** different encoding schemes and selection methods in GAs.

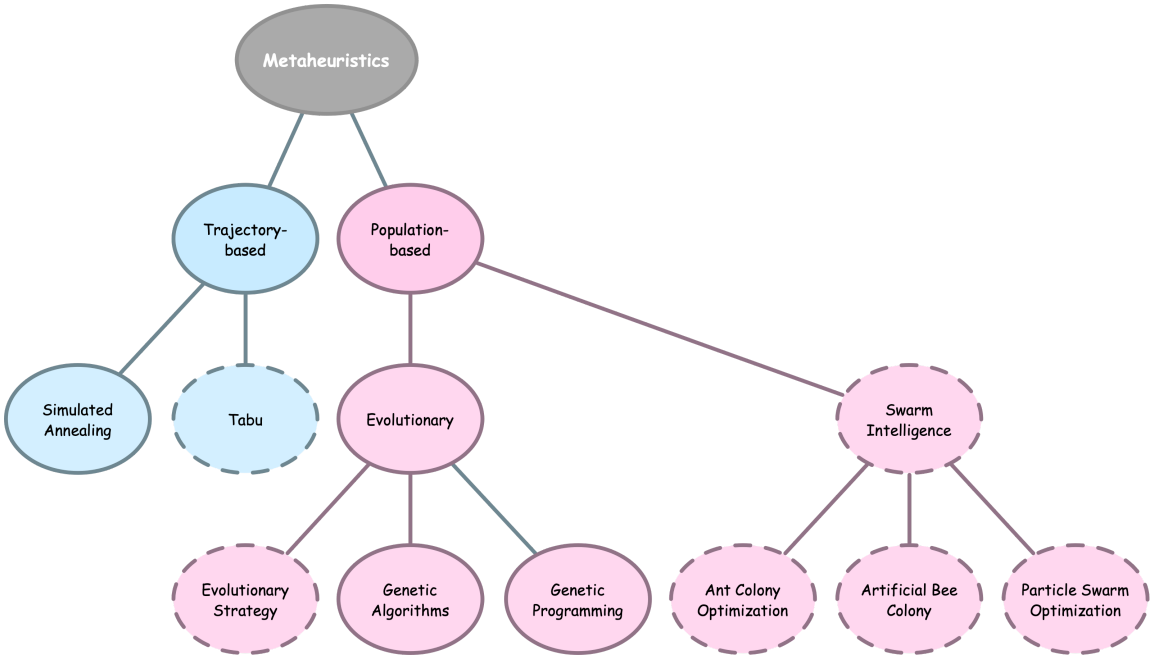
See also: companion Jupyter Notebook.

Genetic algorithms, similar to other topics covered in this course, are a broad and in-depth field of study. They have been explored in numerous specialized works and are the professional focus of some researchers. In this context, we will develop an implementation that is both simple and efficient by utilizing the Numpy library. This approach will be applied to the 0/1 knapsack problem.

Genetic algorithms, although effective in various contexts, have sparked some controversy. Their implementation requires considerable expertise and careful tuning of numerous parameters, which can be time-consuming. Furthermore, some critics argue that the biological metaphor underlying these algorithms complicates their implementation without providing significant added value.

Introduction

Metaheuristics



Simulated annealing draws inspiration from principles in **physics**. In contrast, we now explore a **biologically** inspired strategy.

Simulated Annealing and Hill Climbing are single-trajectory optimization methods: they maintain and update a single candidate solution. In contrast, the algorithm introduced today will operate on a population of solutions.

Definition

> ** (Gil-Rios et al. 2021)**
>
> **Metaheuristics** are higher-level procedures or heuristics designed to **guide the search** for solutions in optimization problems with **large solution spaces**, aiming to find good solutions more efficiently than traditional methods.

Meta heuristics balance **exploitation** and **exploration** to **avoid local optima**, often incorporating **randomness**, **memory**, or **adaptive mechanisms**.

They are adaptable to **diverse optimization problems**.

Definition

A **genetic algorithm** is an evolutionary optimization technique that uses a **population of candidate solutions**, evolving them through **selection**, **crossover**, and **mutation** to iteratively improve towards an “optimal” solution.

Genetic Algorithms

Computer programs that “evolve” in ways that resemble natural selection can solve complex problems even their creators do not fully understand

by John H. Holland

Holland (1992), Holland (1973)

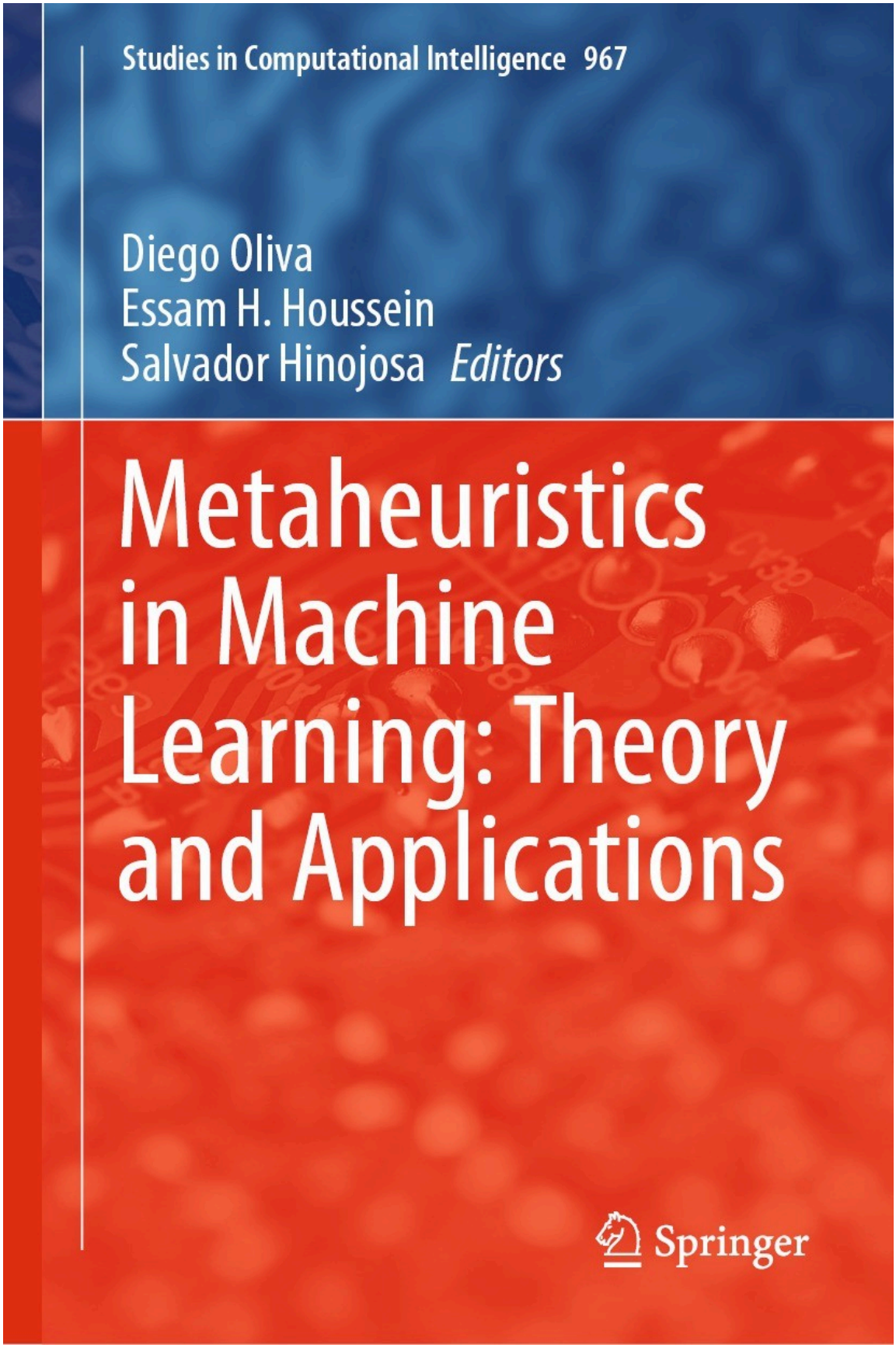
In 1950, Alan Turing introduced the concept of a machine capable of learning by drawing on principles derived from biological evolution. This idea gained prominence in the early 1970s through the seminal contributions of John Holland.

Trends in AI

Applications

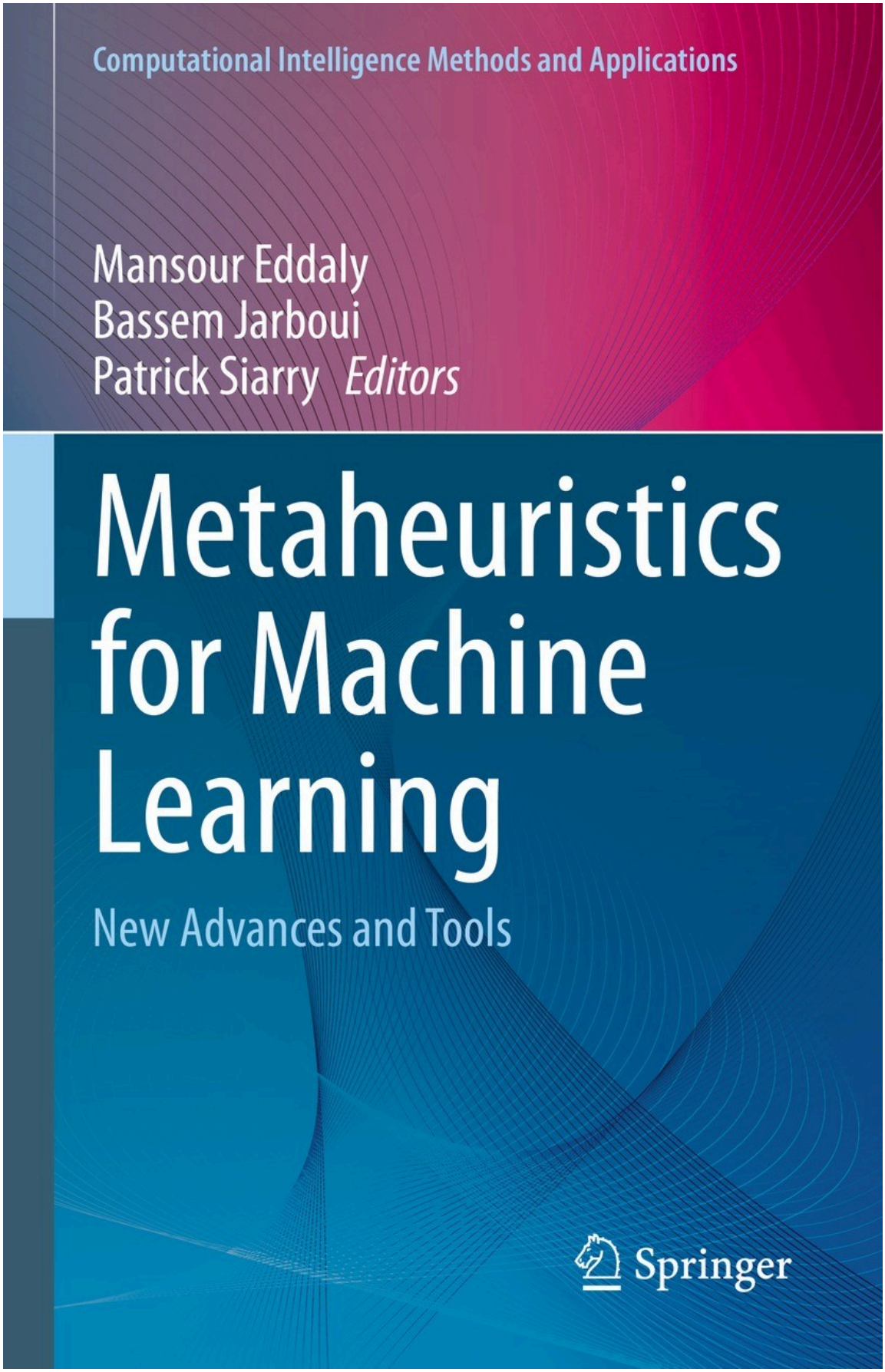
- **Optimization**: Solving complex engineering, logistics, and scheduling problems.
- **Machine Learning**: Feature selection, hyperparameter tuning, and evolving neural network architectures.
- **Robotics**: Path planning, sensor optimization, control strategy development, and robot designs[1].

Applications (continued)



Oliva, Houssein, and Hinojosa (2021)

Springer Link



Eddaly, Jarboui, and Siarry (2023)

Springer Link

Did you know that you can freely access the entire collection of books from Springer? By using a device connected to a uOttawa IP address and visiting Springer Link, you have the ability to download books in either PDF or EPUB format.

From Biology to Genetic Algorithms

> ** (Mayr 1982, 481)**
>
> There is probably no more **original**, more **complex**, and **bolder** concept in the history of ideas than Darwin's mechanistic explanation of adaptation.

> ** (Dennett 1995)**
>
> If I were to give an award for the **single best idea anyone has ever had**, I'd give it to Darwin, ahead of Newton & Einstein and everyone else.

Definition

> ** (Gregory 2009)**
>
> **Natural selection** is a **non-random difference in reproductive output among replicating entities**, often due indirectly to differences in survival in a particular environment, **leading to an increase in the proportion of beneficial, heritable characteristics** within a population from one generation to the next.

Evolution is an **adaptive optimization process**.

Discussion

- What are the essential components?
- What is the structure of the algorithm?

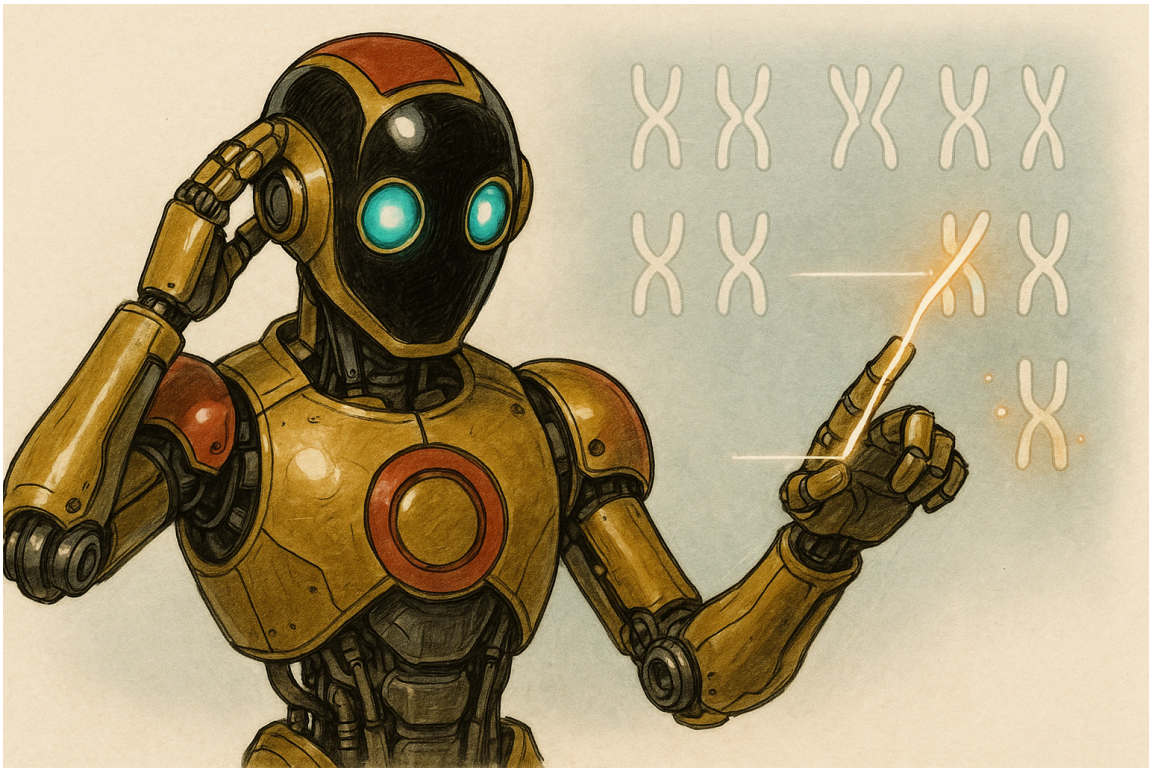


Image generated with OpenAI's DALL·E (2025-11-16).

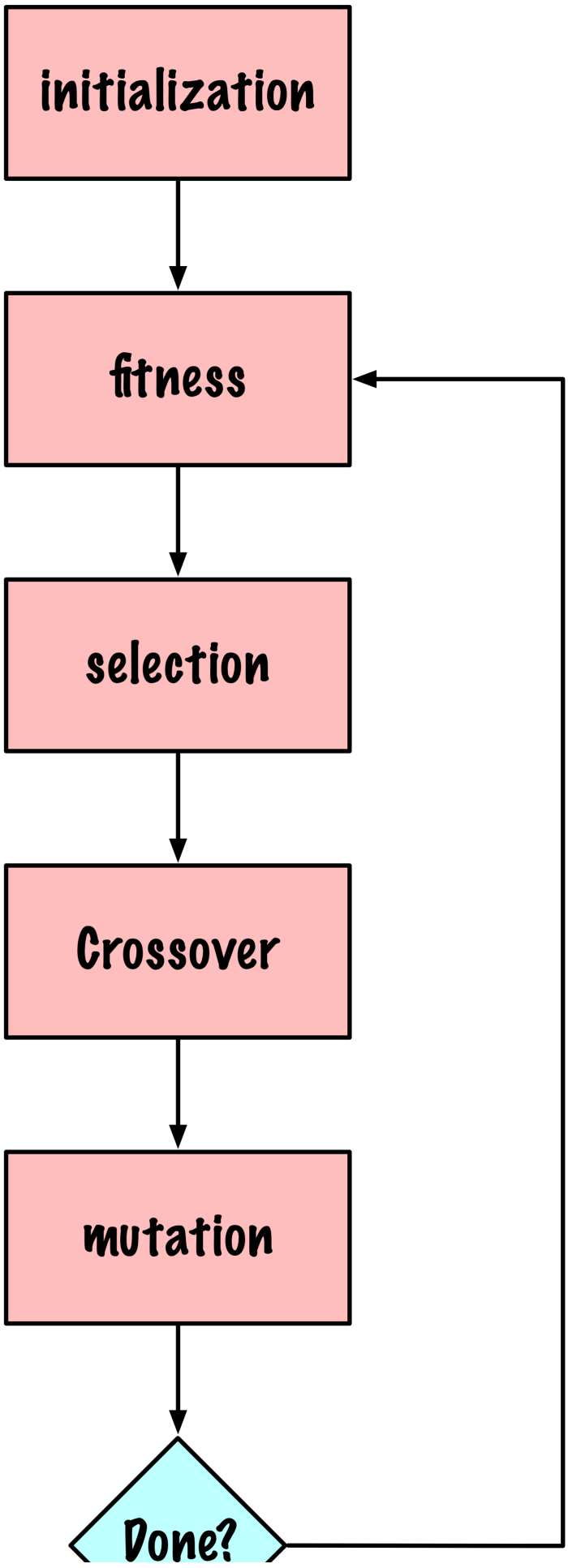
Take a moment to reflect. We have explored various search algorithms, among which the simulated annealing algorithm, a metaheuristic, could be an excellent starting point. If you were to implement a genetic algorithm, what would be its main features? How would the main loop be structured? What auxiliary methods would be necessary? What data structures would be fundamental? What would be its parameters?

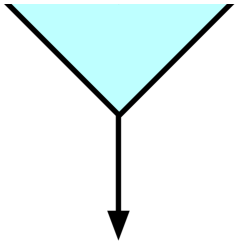
Prompt: "Create a humanoid robot with a vintage-futuristic aesthetic, inspired by the style of Atom: The Beginning. The robot has a smooth helmet-like head with a glossy black visor and two large glowing lenses, a golden-bronze mechanical body with exposed joints and wiring, and red circular accents on the torso. Depict the robot as an intelligent engineer engaged in a genetic algorithm workflow. In front of it, show a floating array of candidate solutions—abstract geometric 'chromosomes' or vector-like shapes—forming an initial population. The robot is actively performing selection, combining two candidate structures with glowing lines to represent cross-over, while subtle distortions or sparks on another candidate indicate mutation.

The composition should look like an upper-body portrait: the robot slightly angled, one hand raised as if manipulating holographic data. The background is light and minimal, reminiscent of a manga cover. Emphasize clarity and soft lighting to highlight metallic textures, glowing eyes, and the dynamic visual metaphor of evolutionary optimization."

Components

Basic GA


initialization
fitness
selection
Crossover
mutation
Done?



1. Initialize **population**
2. Compute the **fitness**
3. Select **individuals** (chromosomes)
4. **Crossover**
5. **Mutation**
6. If not done, **goto** 2

In this context, the term **population** refers to a collection of candidate solutions or states.

Choices

- How to **encode** a candidate solution or state?
- How to **select** candidate solutions?
- How to define the **crossover** operator?
- How to define the **mutation** operator?
- How to calculate the **fitness**?

Problem

0/1 knapsack problem: Given items with defined **weights** and **values**, the objective is to **maximize total value** by selecting items for a knapsack without surpassing a **fixed capacity**. Each item must be either fully **included (1)** or **excluded (0)**.

Attribution: Generated by DALL-E, via ChatGPT (GPT-4), OpenAI, November 10, 2024.

This problem is NP-complete and often solved using dynamic programming for optimal solutions in feasible time.

The image displayed above was generated based on the following prompt: “Illustrate a cartoon girl scout wearing glasses, dressed in a scout uniform featuring brown shorts and hiking boots. She holds a ledger in one hand and a pencil in the other, with a large backpack secured to her back. In front of her, display a variety of items she might choose to pack into her backpack. These items should include a stack of books, a stuffed toy, a cup, assorted snacks, and a pair of binoculars. The scene should evoke a sense of decision-making and exploration.”

Problem (continued)

$$\text{maximize} \sum_{i=1}^{n} x_i \cdot v_i$$

$$\text{subject to} \sum_{i=1}^{n} x_i \cdot w_i \leq W \text{ and } x_i \in \{0, 1\}.$$

where v_i and w_i represent the **value** and the **weight** of item i, respectively, W represents the **fixed maximum weight**, x_i is a binary variable indicating if item i is **included (1)** or **excluded (0)**, and there are n items.

Applications

1. **Finance and Investment**: In portfolio optimization, where each asset has a risk (analogous to weight) and expected return (value), the knapsack framework helps select a set of assets that maximizes return without exceeding a risk threshold.

2. **Resource Allocation**: Common in project management, where resources (budget, personnel, time) need to be allocated across projects or tasks to maximize overall value, taking into account limited availability.

3. **Supply Chain and Logistics**: Used to maximize the value of goods transported within vehicle weight or volume constraints. It can also be applied in warehouse storage, where space is limited, and high-value items are prioritized.

4. **Ad Placement and Marketing**: Used in digital advertising to select the most profitable combination of ads to display within limited space (e.g., website or app banner space), maximizing revenue under size or display constraints.

Greedy Algorithms

> ** (Skiena 2008, 192)**
>
> **Greedy algorithms** make the decision of what to do next by selecting the **best local option** from all available choices **without** regard to the global structure.

Data

[1] This approach has led to the development of **unique**, sometimes **unexpected** robot forms and control strategies that are **well-suited to specific tasks or environments**, often outperforming traditional, human-engineered designs in **adaptability** and **robustness**.

In [1]:
```python
import numpy as np

# Sample data

values = np.array([
    360, 83, 59, 130, 431, 67, 230, 52, 93, 125, 670, 892, 600, 38, 48, 147,
    78, 256, 63, 17, 120, 164, 432, 35, 92, 110, 22, 42, 50, 323, 514, 28,
    87, 73, 78, 15, 26, 78, 210, 36, 85, 189, 274, 43, 33, 10, 19, 389, 276,
    312])

weights = np.array([
    7, 0, 30, 22, 80, 94, 11, 81, 70, 64, 59, 18, 0, 36, 3, 8, 15, 42, 9, 0,
    42, 47, 52, 32, 26, 48, 55, 6, 29, 84, 2, 4, 18, 56, 7, 29, 93, 44, 71,
    3, 86, 66, 31, 65, 0, 79, 20, 65, 52, 13])

capacity = 850
```

Data from Google OR-Tools.

Ascending order of weight

In [2]:
```python
def greedy_knapsack_weight(weights, values, capacity):

    num_items = len(weights)

    # Create a list of items with their values and original indices
    items = list(zip(weights, values, range(num_items)))

    # Sort items by weight in increasing order
    items.sort()

    total_weight, total_value = 0,0
    solution = np.zeros(num_items, dtype=int)

    # Select items based on the sorted order
    for w, v, idx in items:
        if total_weight + w <= capacity:
            solution[idx] = 1
            total_weight += w
            total_value += v
        else:
            break  # Skip items that would exceed the capacity

    return solution, total_value, total_weight
```

Is this strategy **relevant**?

- What is the time complexity of this method?
  - $\mathcal{O}(n \log n)$ due to sorting.
- Is this strategy relevant?
  - It overlooks value, the critical factor for optimization.

`greedy_knapsack_weight`

In [3]:
```
solution, total_value, total_weight = greedy_knapsack_weight(weights, values

print(f"Solution: {solution}")
print(f"Value: {total_value}")
print(f"Weight: {total_weight}")
```

```
Solution: [1 1 1 1 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
1 1 1 0
 1 0 1 0 0 1 0 1 0 1 0 1 1]
Value: 5891
Weight: 817
```

Descending value order

In [4]:
```
def greedy_knapsack_value(weights, values, capacity):

    num_items = len(weights)

    # Create a list of items with their values and original indices
    items = list(zip(values, weights, range(num_items)))

    # Sort items by value in decreasing order
    items.sort(reverse=True)

    total_weight = 0
    total_value = 0
    solution = np.zeros(num_items, dtype=int)

    # Select items based on the sorted order
    for v, w, idx in items:
        if total_weight + w <= capacity:
            solution[idx] = 1
            total_weight += w
            total_value += v

    return solution, total_value, total_weight
```

- What is the time complexity of this method?
  - $\mathcal{O}(n \log n)$ due to sorting.
- Is this strategy relevant?
  - Emphasizes maximizing value but does not guarantee an optimal solution.

`greedy_knapsack_value`

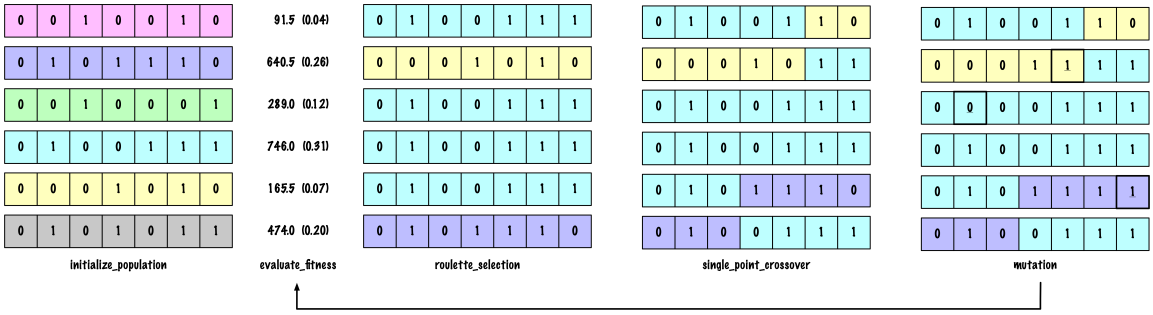
In [5]:
```
solution, total_value, total_weight = greedy_knapsack_value(weights, values,

print(f"Solution: {solution}")
```

```
print(f"Value: {total_value}")
print(f"Weight: {total_weight}")
```

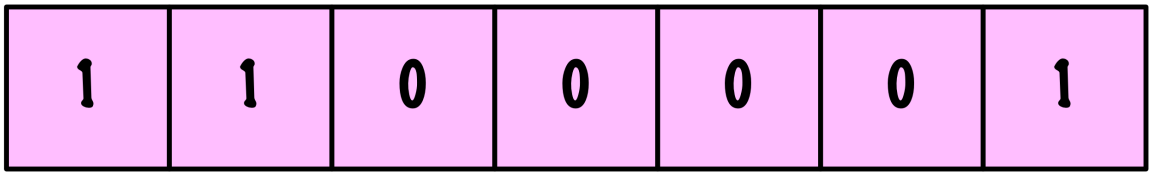
```
Solution: [1 1 0 1 1 0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 0 0
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7339
Weight: 849
```

GA Implementation

Overview



Encoding



The most **straightforward representation** for each **chromosome** (**state**) is an array of size n, where n represents the number of items. Each element i in the array indicates whether item i is **included (1)** or **excluded (0)**.

Population

In [6]:
```
def initialize_population(pop_size, num_items):

    """
    Initialize the population with random binary strings.

    Args:
        pop_size (int): Number of individuals in the population.
        num_items (int): Number of items in the knapsack problem.

    Returns:
        np.ndarray: Initialized population.
    """
```

```
    return np.random.randint(2, size=(pop_size, num_items))
```

How to choose the **population size (number of individuals/chromosomes)**?

Problem specificity: Adjust the population size based on the complexity of the problem. More complex problems may require a larger population to effectively explore the solution space.

Available resources: Consider the available computing resources, as a larger population size increases memory and computational time requirements.

Balance between diversity and convergence: A larger population can maintain higher genetic diversity, which helps avoid local optima, but may also slow convergence. Find a balance suitable for your problem.

Guidelines: For many problems, an empirical rule is to start with a population size between 50 and 100, then adjust based on observed performance.

Adaptive approaches: Consider using adaptive methods that dynamically adjust the population size based on the algorithm's progress.

Data

In [7]:
```
pop_size_small = 12
num_items_small = 10
values_small = values[:num_items_small]
weights_small = weights[:num_items_small]
capacity_small = 125
```

Let's define a small problem to illustrate the algorithm.

Population

In [8]:
```
population = initialize_population(pop_size_small, num_items_small)
population
```

```
array([[1, 0, 1, 0, 0, 1, 1, 0, 1, 0],
       [1, 1, 0, 1, 1, 1, 0, 0, 1, 1],
       [1, 1, 1, 0, 0, 0, 0, 0, 1, 0],
       [1, 0, 0, 0, 1, 0, 1, 1, 1, 1],
       [1, 1, 0, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 0, 1, 1, 1, 0, 0, 1],
       [1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 0, 0, 1],
       [0, 1, 0, 0, 1, 1, 1, 1, 0, 1],
       [0, 1, 0, 1, 1, 0, 1, 0, 1, 1],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 1, 1, 0, 1, 0, 1, 1, 0, 1]])
```

Population

The proposed method for initializing the population presents a **problem**.

Can you identify *what it is?*

The initial population is generated entirely at random without considering feasibility.

This can result in most individuals being infeasible (exceeding capacity), especially for larger problems, slowing down convergence.

Suggestion: Introduce a feasibility check during initialization or generate initial solutions that are more likely to be feasible.

Fitness

In [9]:
```python
def evaluate_fitness(population, weights, values, capacity, penalty_factor=1

    total_weights = np.dot(population, weights)

    total_values = np.dot(population, values)

    penalties = penalty_factor * np.maximum(0, total_weights - capacity)

    fitness = total_values - penalties

    return fitness
```

Can you identify a possible **issue** with this fitness function?

First, notice the extensive use of numpy. The total weight for each chromosome of our population is calculated with a single expression. Likewise for total values.

The fitness function employs a smoother penalization strategy by deducting a penalty proportional to the excess weight, thereby distinguishing between solutions that exceed the weight limit.

Alternatively, one could assign a fitness value of zero to any solution that exceeds the knapsack's capacity. However, this stringent penalization could result in a significant portion, or even all, of the population having zero fitness, particularly in the initial generations. This scenario poses challenges for selection mechanisms, such as roulette wheel selection, which rely on non-zero fitness values to calculate selection probabilities, potentially leading to division by zero errors.

The proposed strategy, which applies penalties to overweight solutions, may lead the genetic algorithm to generate infeasible solutions. Such instances require external

handling by restarting the genetic algorithm whenever an infeasible solution is encountered.

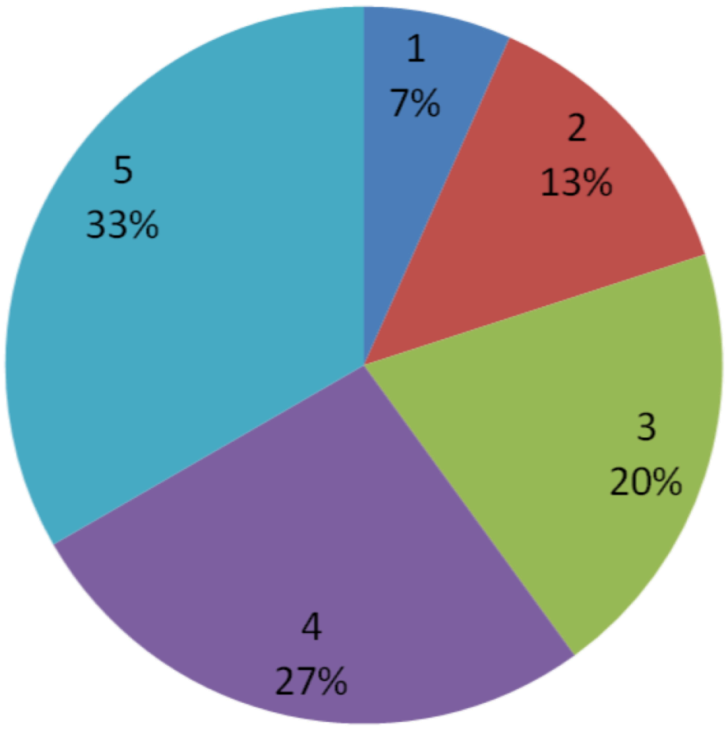
Fitness

In [10]:
```
fitness = evaluate_fitness(population, weights_small, values_small, capacity
fitness
```

```
array([ 765.5, 1183. ,  595. , 1197. , 1163. , 1230.5,  427. , 1393.5,
        885.5, 1031. ,   67. ,  909.5])
```

Roulette Wheel Selection

Roulette wheel selection is a stochastic selection method where the probability of selecting an individual is proportional to its fitness relative to the rest of the population.



Individual	Fitness
1	1.0
2	2.0
3	3.0
4	4.0
5	5.0

Fig. 2. Roulette wheel selection example

Attribution: Santos Amorim et al. (2012)

What is the fundamental objective of this selection method? Why not simply choose the individuals with the highest fitness?

The fundamental objective is to maintain a diverse population.

Advantages

1. **Fitness Proportionality**
    - **Simple Implementation**: The method is straightforward to implement, requiring the calculation of cumulative probabilities based on fitness values.
    - **All Individuals Have a Chance**: Even individuals with lower fitness have a non-zero probability of being selected, maintaining genetic diversity.
2. **Selection Pressure Adjusted by Fitness**
    - **Dynamic Pressure**: Selection pressure naturally adjusts based on the fitness distribution of the population.
    - **Encourages High-Fitness Individuals**: Individuals with higher fitness are more likely to be selected, promoting the propagation of advantageous traits.

Disadvantages

1. **Sensitivity to Fitness Scaling**
    - **Negative Fitness Values**: If fitness values are negative or zero (which can happen when penalties are applied), the method becomes problematic as probabilities cannot be negative or zero.
    - **Scaling Issues**: When fitness values are very close together or have a small range, the selection probabilities become nearly uniform, reducing selection pressure.
2. **Premature Convergence Risk**
    - **Dominance of High-Fitness Individuals**: If a few individuals have significantly higher fitness, they may dominate the selection process, leading to reduced diversity and premature convergence to suboptimal solutions.
3. **Computational Overhead**
    - **Normalization Required**: Fitness values need to be normalized to probabilities, which adds computational steps, especially if adjustments are needed for negative fitness values.

`roulette_selection`

In [11]:
```python
def roulette_selection(population, fitness):

    # Adjust fitness to be non-negative
    min_fitness = np.min(fitness)
    adjusted_fitness = fitness - min_fitness + 1e-6  # small epsilon to avoi

    total_fitness = np.sum(adjusted_fitness)
    probabilities = adjusted_fitness / total_fitness
```

```
    pop_size = population.shape[0]
    selected_indices = np.random.choice(pop_size, size=pop_size, p=probabili

    return population[selected_indices]
```

In your own words, what does `roulette_selection` do?

A small value is added to the adjusted fitness values to avoid dividing by 0.

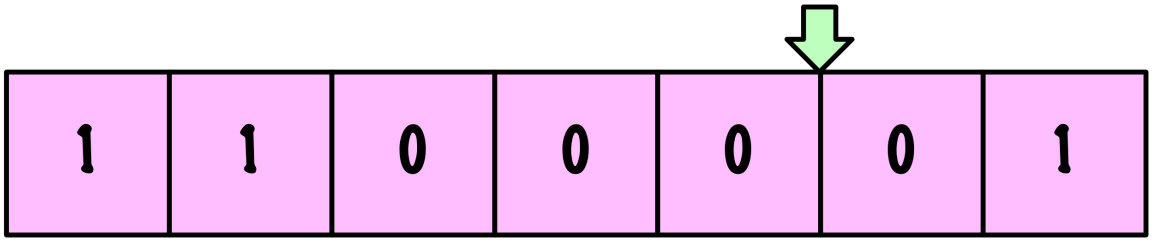
Returns a population of the same size. The fittest individuals should now be more represented.

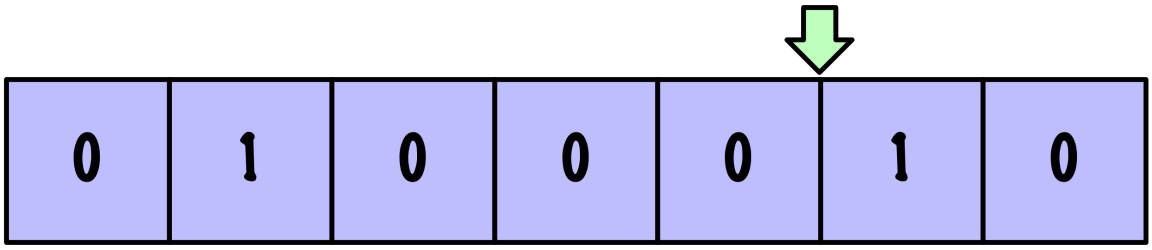
`roulette_selection`

In [12]:
```
offspring = roulette_selection(population, fitness)
offspring
```

```
array([[0, 1, 0, 0, 1, 1, 1, 1, 0, 1],
       [0, 1, 0, 1, 1, 0, 1, 0, 1, 1],
       [0, 1, 1, 0, 1, 0, 1, 1, 0, 1],
       [1, 1, 0, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 0, 0, 1, 1, 1, 0, 0, 1],
       [0, 1, 0, 1, 1, 0, 1, 0, 1, 1],
       [1, 1, 0, 1, 1, 0, 0, 0, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 0, 0, 1],
       [1, 1, 0, 1, 1, 1, 0, 0, 1, 1],
       [1, 1, 0, 0, 1, 1, 1, 0, 0, 1],
       [1, 1, 1, 1, 1, 1, 1, 0, 0, 1],
       [1, 1, 0, 1, 1, 1, 0, 0, 1, 1]])
```

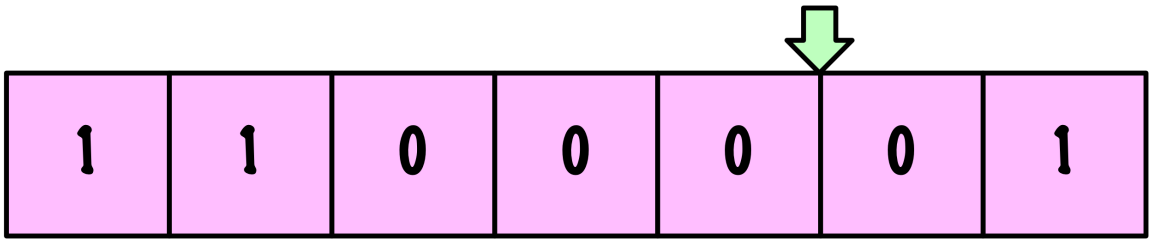
Crossover

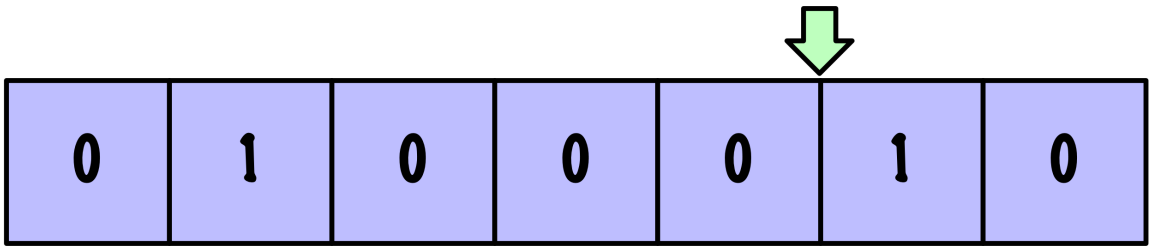




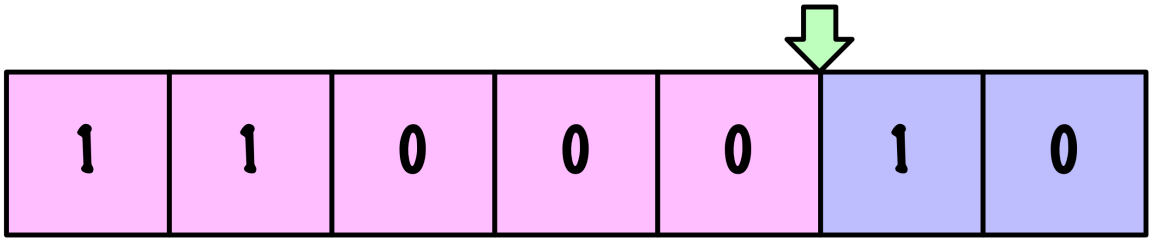
In the **single crossover** method, a crossover point is **randomly selected**. The resulting child 1 is formed by combining the **prefix** of parent 1 with the **suffix** of parent 2, while child 2 is created by joining the suffix of parent 1 with the prefix of parent 2.

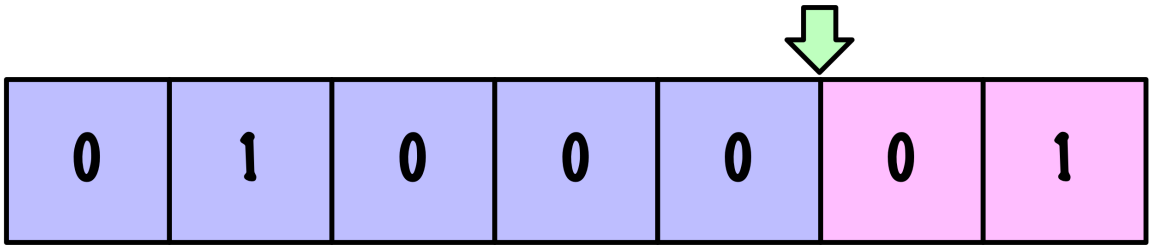
Crossover





Parents





Offspring

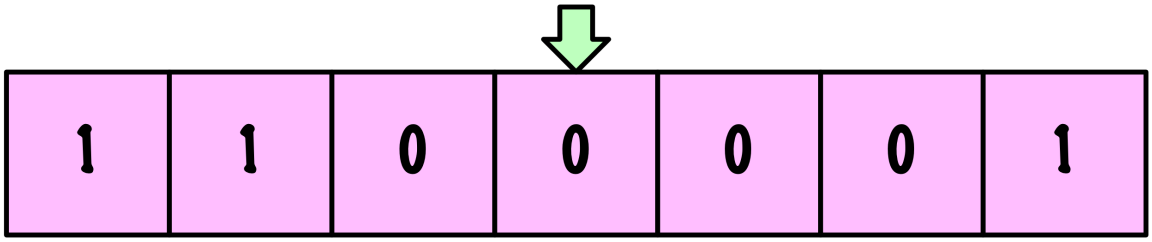
Two **parent individuals** produce two **offspring**.

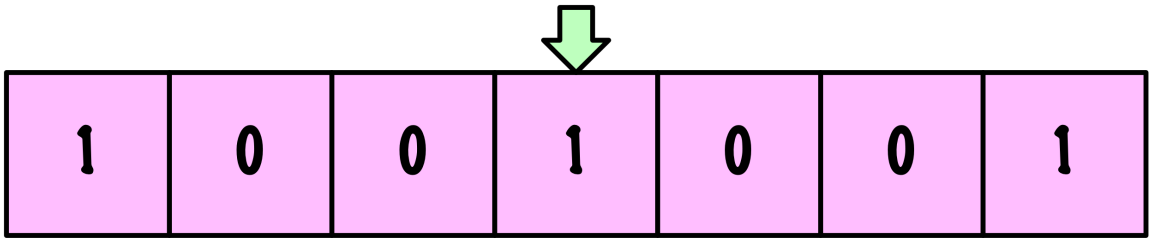
Crossover

In [13]:
```
def single_point_crossover(parents, crossover_rate):

    num_parents, num_genes = parents.shape
    np.random.shuffle(parents)
    offspring = []

    for i in range(0, num_parents, 2):

        parent1 = parents[i]
        parent2 = parents[i+1 if i+1 < num_parents else 0]

        child1 = parent1.copy()
        child2 = parent2.copy()

        if np.random.rand() < crossover_rate:
            point = np.random.randint(1, num_genes)  # Crossover point
            child1[:point], child2[:point] = parent2[:point], parent1[:point

        offspring.append(child1)
        offspring.append(child2)

    return np.array(offspring)
```

Controlled by a user-specified parameter, `crossover_rate`, typical value 0.8.

Mutation





In this basic scenario, a **mutation** involves "flipping" a bit.

Mutate

In [14]:
```
def mutation(offspring, mutation_rate):

    num_offspring, num_genes = offspring.shape

    mutation_matrix = np.random.rand(num_offspring, num_genes) < mutation_ra

    offspring[mutation_matrix] = 1 - offspring[mutation_matrix]

    return offspring
```

A user-defined **mutation rate** determines whether a **position** undergoes "mutation."

Clarification

In [15]:
```
np.random.seed(42)

offspring = initialize_population(4, 10)

num_offspring, num_genes = offspring.shape

print("Offspring:")
print(offspring)
```

```
Offspring:
[[0 1 0 0 0 1 0 0 0 1]
 [0 0 0 0 1 0 1 1 1 0]
 [1 0 1 1 1 1 1 1 1 1]
 [0 0 1 1 1 0 1 0 0 0]]
```

In [16]:
```
mutation_rate = 0.05
mutation_matrix = np.random.rand(num_offspring, num_genes) < mutation_rate

print("Mutation matrix:")
print(mutation_matrix)
```

```
Mutation matrix:
[[False False False False False False False False False  True]
 [False False False False False False False False False False]
 [False False  True False False False False False False False]
 [False False False False False False False False  True False]]
```

In [17]:
```
print("offspring[mutation_matrix]:")
print(offspring[mutation_matrix])
```

```
offspring[mutation_matrix]:
[1 1 0]
```

In [18]:
```
print("1 - offspring[mutation_matrix]:")
print(1 - offspring[mutation_matrix])
```

```
1 - offspring[mutation_matrix]:
[0 0 1]
```

In [19]:
```
offspring[mutation_matrix] = 1 - offspring[mutation_matrix]

print("Mutated offstring:")
print(offspring)
```

```
Mutated offstring:
[[0 1 0 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 1 1 1 0]
 [1 0 0 1 1 1 1 1 1 1]
 [0 0 1 1 1 0 1 0 1 0]]
```

This implementation makes extensive use of Numpy.

Elitism

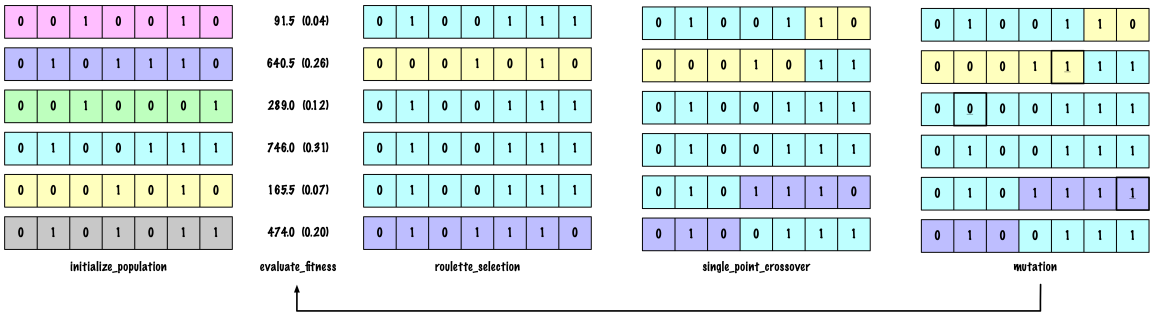
Elitism in genetic algorithms is a strategy where a subset of the fittest individuals from the current generation is directly carried over to the next generation.

This approach ensures that the best solutions are preserved throughout the evolutionary process, enhancing convergence speed and maintaining high-quality solutions within the population.

Elitism

In [20]:
```python
def elitism(population, fitness, elite_size):

    elite_indices = np.argsort(fitness)[-elite_size:]  # Get indices of top

    elites = population[elite_indices]

    return elites
```

Overview



The figure above presents several noteworthy elements that can fuel discussion. The initial population, fitness values, and the population resulting from roulette wheel selection were generated using the code we previously introduced. It is notable that individuals with fitness values of 91.5, 289.0, and 474.0 do not appear in the population obtained after roulette wheel selection. While the absence of the individual with a value of 91.5 might seem expected, the absence of the individual with a value of 474.0 could be surprising. However, this can be explained by the probabilistic nature of the algorithm. Additionally, it is worth noting that the population size is limited, which influences the results. Furthermore, the individual with the highest fitness value was selected multiple times.

The numbers in parentheses indicate the probabilities used by the roulette wheel selection algorithm.

Genetic Algorithm (Version 1)

In [21]:
```python
def genetic_algorithm(weights, values, capacity, pop_size=100,
                      num_generations=100, crossover_rate=0.8,
                      mutation_rate=0.05, elite_percent=0.02):

    num_items = len(weights)
    elite_size = max(1, int(pop_size * elite_percent))
    population = initialize_population(pop_size, num_items)

    for generation in range(num_generations):

        fitness = evaluate_fitness(population, weights, values, capacity)

        # Elitism
```

```
        elites = elitism(population, fitness, elite_size)

        # Selection
        parents = roulette_selection(population, fitness)

        # Crossover
        offspring = single_point_crossover(parents, crossover_rate)

        # Mutation
        offspring = mutation(offspring, mutation_rate)

        # Create new population
        population = np.vstack((elites, offspring))

        # Ensure population size
        if population.shape[0] > pop_size:
            population = population[:pop_size]
        elif population.shape[0] < pop_size:
            # Add random individuals to fill population
            num_new_individuals = pop_size - population.shape[0]
            new_individuals = initialize_population(num_new_individuals, num
            population = np.vstack((population, new_individuals))

    # After all generations, return the best solution
    fitness = evaluate_fitness(population, weights, values, capacity)
    best_index = np.argmax(fitness)
    best_solution = population[best_index]
    best_value = np.dot(best_solution, values)
    best_weight = np.dot(best_solution, weights)

    return best_solution, best_value, best_weight
```

Run

In [22]:
```
np.random.seed(13)

solution, total_value, total_weight = genetic_algorithm(weights, values, cap

print(f"Solution: {solution}")
print(f"Value: {total_value}")
print(f"Weight: {total_weight}")
```

```
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 1
1 1 0 0
 1 0 1 0 0 1 0 1 0 0 1 1 1]
Value: 7357
Weight: 848
```

Genetic Algorithm (Version 1.1)

In [23]:
```
def genetic_algorithm(weights, values, capacity, pop_size=100,
                      num_generations=100, crossover_rate=0.8,
                      mutation_rate=0.05, elite_percent=0.02):
```

```
    num_items = len(weights)
    elite_size = max(1, int(pop_size * elite_percent))
    population = initialize_population(pop_size, num_items)

    average_fitness_history = []
    best_fitness_history = []

    for generation in range(num_generations):

        fitness = evaluate_fitness(population, weights, values, capacity)

        # Track average and best fitness
        average_fitness = np.mean(fitness)
        best_fitness = np.max(fitness)
        average_fitness_history.append(average_fitness)
        best_fitness_history.append(best_fitness)

        # Elitism
        elites = elitism(population, fitness, elite_size)

        # Selection
        parents = roulette_selection(population, fitness)

        # Crossover
        offspring = single_point_crossover(parents, crossover_rate)

        # Mutation
        offspring = mutation(offspring, mutation_rate)

        # Create new population
        population = np.vstack((elites, offspring))

        # Ensure population size
        if population.shape[0] > pop_size:
            population = population[:pop_size]
        elif population.shape[0] < pop_size:
            # Add random individuals to fill population
            num_new_individuals = pop_size - population.shape[0]
            new_individuals = initialize_population(num_new_individuals, num
            population = np.vstack((population, new_individuals))

    # After all generations, return the best solution
    fitness = evaluate_fitness(population, weights, values, capacity)
    best_index = np.argmax(fitness)
    best_solution = population[best_index]
    best_value = np.dot(best_solution, values)
    best_weight = np.dot(best_solution, weights)

    return best_solution, best_value, best_weight, average_fitness_history,
```

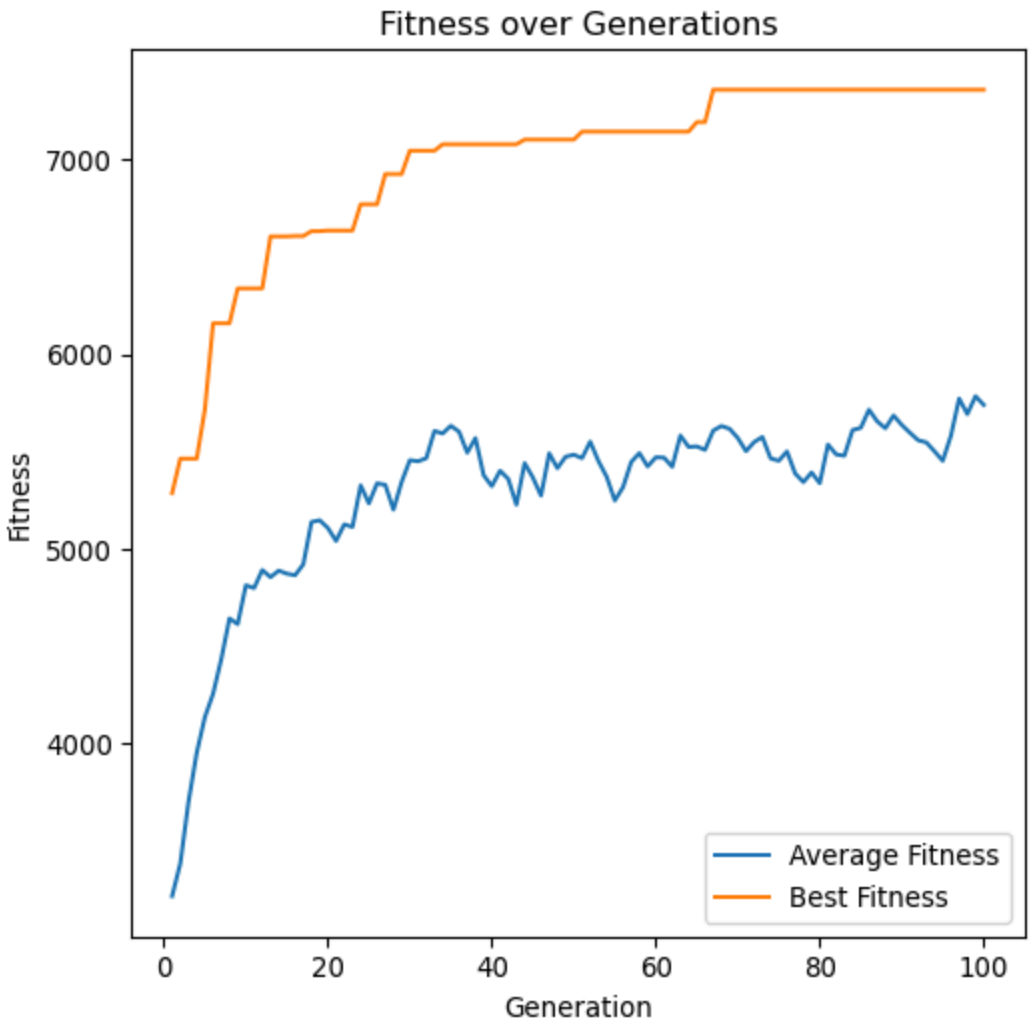
Plot

In [24]:
```python
import matplotlib.pyplot as plt

def plot_fitness_over_generations(avg_fitness_history, best_fitness_history)

  generations = range(1, len(avg_fitness_history) + 1)

  plt.figure(figsize=(6, 6))

  plt.plot(generations, avg_fitness_history, label='Average Fitness')
  plt.plot(generations, best_fitness_history, label='Best Fitness')
  plt.xlabel('Generation')
  plt.ylabel('Fitness')
  plt.title('Fitness over Generations')
  plt.legend()
```

Run

In [25]:
```python
np.random.seed(13)

solution, total_value, total_weight, avg_fitness_history, best_fitness_histo

print(f"Solution: {solution}")
print(f"Value: {total_value}")
print(f"Weight: {total_weight}")

plot_fitness_over_generations(avg_fitness_history, best_fitness_history)
```

```
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 1
1 1 0 0
 1 0 1 0 0 1 0 1 0 0 1 1 1]
Value: 7357
Weight: 848
```



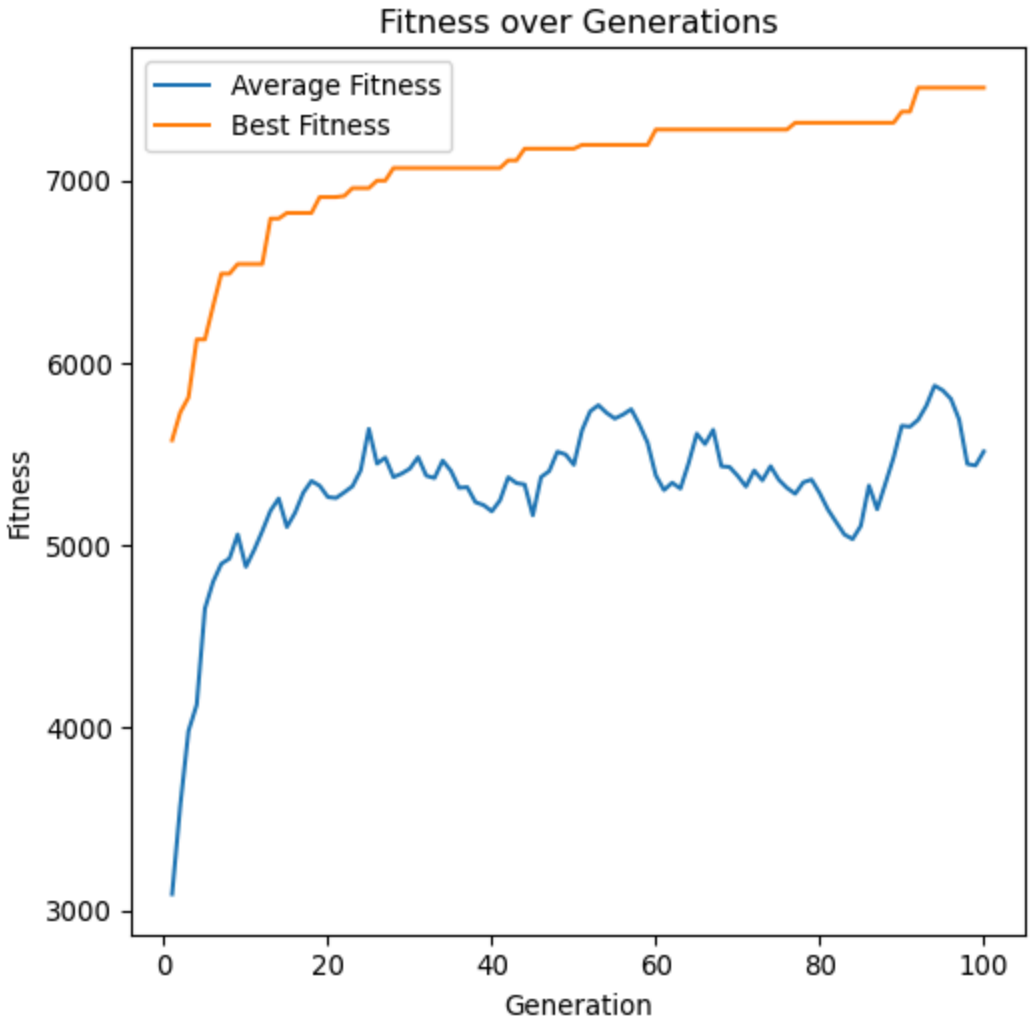
Run

In [26]:

```
np.random.seed(42)

best_value = -1
best_weight = -1
best_solution, best_averages, best_bests = None,  None, None

for i in range(100):

  solution, total_value, total_weight, avg_fitness_history, best_fitness_his

  if total_value > best_value and total_weight <= capacity:
    best_value = total_value
    best_weight = total_weight
    best_solution = solution
    best_averages = avg_fitness_history
    best_bests = best_fitness_history


print(f"Solution: {best_solution}")
print(f"Value: {best_value}")
print(f"Weight: {best_weight}")
```

```
plot_fitness_over_generations(best_averages, best_bests)
```

```
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 0 1
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7506
Weight: 846
```



Using Google OR Tools

In [27]:
```
# https://developers.google.com/optimization/pack/knapsack

from ortools.algorithms.python import knapsack_solver

def solve_using_ortools(values, weights, capacity):

  weights = [weights]

  # Create the solver

  solver = knapsack_solver.KnapsackSolver(
      knapsack_solver.SolverType.KNAPSACK_MULTIDIMENSION_BRANCH_AND_BOUND_SC
      "KnapsackExample",
```

```
    )

    solver.init(values, weights, [capacity])

    computed_value = solver.solve()

    packed_items = []
    packed_weights = []
    total_weight = 0
    print("Total value =", computed_value)
    for i in range(len(values)):
        if solver.best_solution_contains(i):
            packed_items.append(i)
            packed_weights.append(int(weights[0][i]))
            total_weight += int(weights[0][i])
    print("Total weight:", total_weight)
    print("Packed items:", packed_items)
    print("Packed_weights:", packed_weights)
```

`solve_using_ortools`

In [28]:
```
solve_using_ortools(values, weights, capacity)
```

```
Total value = 7534
Total weight: 850
Packed items: [0, 1, 3, 4, 6, 10, 11, 12, 14, 15, 16, 17, 18, 19, 21, 22, 2
4, 27, 28, 29, 30, 31, 32, 34, 38, 39, 41, 42, 44, 47, 48, 49]
Packed_weights: [7, 0, 22, 80, 11, 59, 18, 0, 3, 8, 15, 42, 9, 0, 47, 52, 2
6, 6, 29, 84, 2, 4, 18, 7, 71, 3, 66, 31, 0, 65, 52, 13]
```

Aspect	Branch & Bound	Genetic Algorithm
Solution quality	Optimal	High-quality, but typically approximate
Runtime	Fast for moderate (n), but can blow up	Usually stable; can handle large (n)
Scalability	Poor for large (n)	Good; parallel-friendly
Deterministic	Yes	No (stochastic)
Hyperparameters	None	Many
Use case	When optimality is required	When approximate solutions are acceptable, or (n) is large

Discussion

Encoding Schemes

- **Binary encoding** is commonly used.

- **Permutation encoding**. Typically used with problems such as “Travelling Salesman Problem (TSP)” and N-Queens.
- **Values encoding**. In this encoding, integer, real, or character values are used. Example: learning the parameters of a polynomical in a regression problem.

In the context of binary encoding, it specifically refers to the numerical representation in base two. For instance, to determine the optimal value of the function $f(x) = x + |\sin(32x)|$ in the interval $0 \leq x < \pi$, we use a population of numbers represented in binary form.

Selection

Tournament selection involves randomly selecting a subset of individuals (a tournament) from the population and then selecting the best individual from this subset to be a parent. The process is repeated until the required number of parents is selected.

This method balances between **exploration** and **exploitation**. It allows for controlling selection pressure by varying the tournament size. **Larger tournaments increase selection pressure**, favoring the fittest individuals more strongly.

Advantages

1. **Robustness to Fitness Scaling**
    - **Handles Negative Fitness Values**: The method relies on relative ranking within the tournament rather than absolute fitness values, making it robust to negative or zero fitness values.
    - **Less Sensitive to Fitness Distribution**: Since selection is based on comparison rather than proportion, it performs well even when fitness values are close together.
2. **Adjustable Selection Pressure**
    - **Control via Tournament Size**: The selection pressure can be easily adjusted by changing the tournament size:
        - **Larger Tournaments**: Higher selection pressure, as there is a greater chance of selecting the best individuals.
        - **Smaller Tournaments**: Lower selection pressure, promoting diversity.
    - **Simplicity**: Easy to implement without the need for fitness normalization.
3. **Maintains Diversity**
    - **Avoids Premature Convergence**: By not solely focusing on high-fitness individuals, it maintains genetic diversity within the population.

Disadvantages

1. **Stochastic Nature**
    - **Randomness in Selection**: The selection process is more stochastic, and the best individual in the population might not be selected if not included in a tournament.
    - **Possible Slow Convergence**: If the tournament size is too small, the algorithm may converge slowly due to low selection pressure.
2. **Parameter Dependence**
    - **Requires Tuning**: The performance is sensitive to the choice of tournament size, which may require experimentation to optimize.
3. **Computational Considerations**
    - **Multiple Comparisons**: Requires random sampling and comparisons, which might be computationally more intensive for very large populations, though generally negligible.

`tournament_selection`

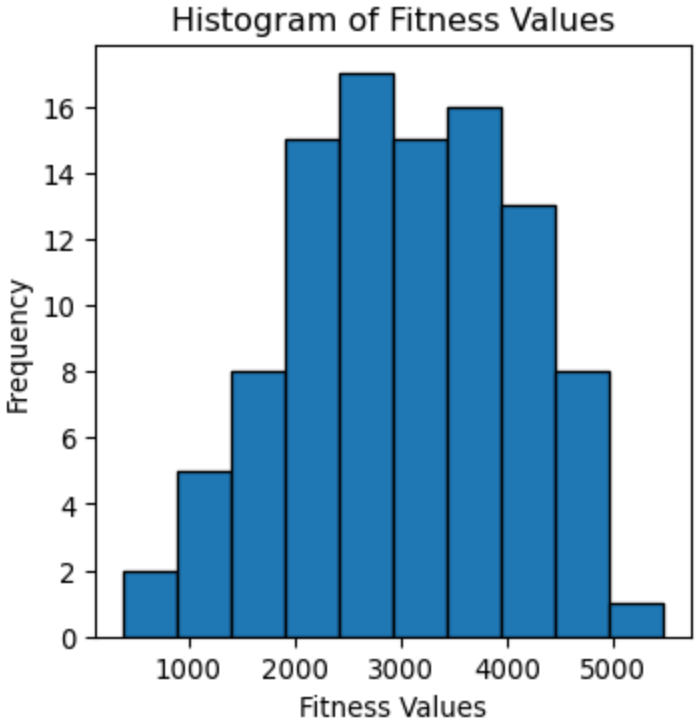
In [29]:
```
def tournament_selection(population, fitness, tournament_size):
    pop_size = population.shape[0]
    selected_indices = []
    for _ in range(pop_size):
        participants = np.random.choice(pop_size, tournament_size, replace=F
        best = participants[np.argmax(fitness[participants])]
        selected_indices.append(best)
    return population[selected_indices]
```

Typical values for `tournament_size` are 2, 3, 4, or 5.

Discussion

In [30]:
```
np.random.seed(27)
pop_size, num_items = 100, 50
population = initialize_population(pop_size, num_items)
fitness = evaluate_fitness(population, weights, values, capacity)
plt.figure(figsize=(4, 4))
```

```
plt.hist(fitness, bins=10, edgecolor='black')
plt.xlabel('Fitness Values')
plt.ylabel('Frequency')
plt.title('Histogram of Fitness Values')
plt.show()

for tournament_size in [2, 4, 8, pop_size]:
  participants = np.random.choice(pop_size, tournament_size, replace=False)
  print(f"tournament_size: {tournament_size}, fitness: {max(fitness[particip
```




```
tournament_size: 2, fitness: 1985
tournament_size: 4, fitness: 2933
tournament_size: 8, fitness: 3966
tournament_size: 100, fitness: 5478
```

- Smaller k (e.g., $k = 2$): Low selection pressure promotes diversity and exploration, useful in early generations.
- Larger k (e.g., $k = 5$): High selection pressure accelerates convergence, emphasizing exploitation, ideal in later generations.

Genetic Algorithm (Version 2)

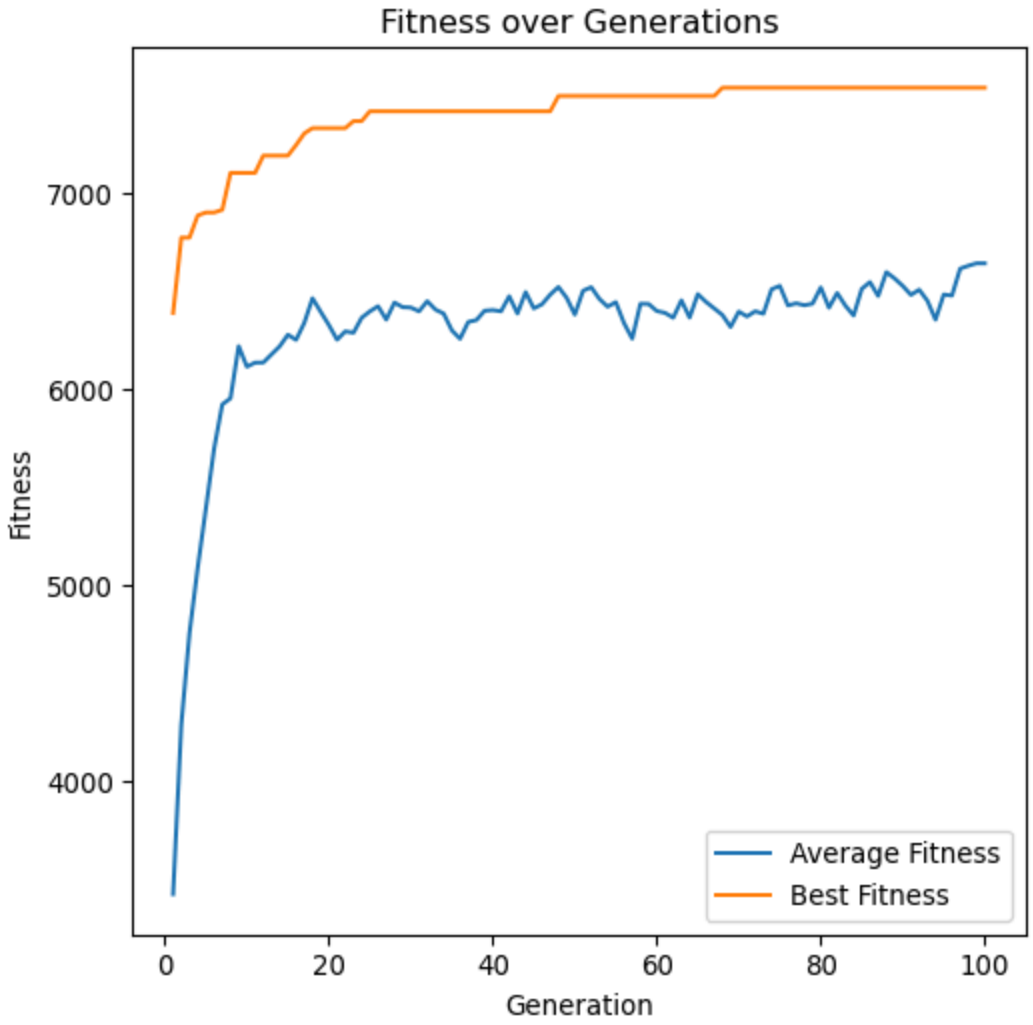
In [31]:
```
def genetic_algorithm(weights, values, capacity, pop_size=100,
                      num_generations=100, crossover_rate=0.8,
                      mutation_rate=0.05, elite_percent=0.02,
                      selection_type='tournament', tournament_size=3):

    num_items = len(weights)
    elite_size = max(1, int(pop_size * elite_percent))
    population = initialize_population(pop_size, num_items)
```

```
    average_fitness_history = []
    best_fitness_history = []

    for generation in range(num_generations):

        fitness = evaluate_fitness(population, weights, values, capacity)

        # Track average and best fitness
        average_fitness = np.mean(fitness)
        best_fitness = np.max(fitness)
        average_fitness_history.append(average_fitness)
        best_fitness_history.append(best_fitness)

        # Elitism
        elites = elitism(population, fitness, elite_size)

        # Selection
        if selection_type == 'tournament':
            parents = tournament_selection(population, fitness, tournament_s
        elif selection_type == 'roulette':
            parents = roulette_selection(population, fitness)
        else:
            raise ValueError("Invalid selection type")

        # Crossover
        offspring = single_point_crossover(parents, crossover_rate)

        # Mutation
        offspring = mutation(offspring, mutation_rate)

        # Create new population
        population = np.vstack((elites, offspring))

        # Ensure population size
        if population.shape[0] > pop_size:
            population = population[:pop_size]
        elif population.shape[0] < pop_size:
            # Add random individuals to fill population
            num_new_individuals = pop_size - population.shape[0]
            new_individuals = initialize_population(num_new_individuals, num
            population = np.vstack((population, new_individuals))

    # After all generations, return the best solution
    fitness = evaluate_fitness(population, weights, values, capacity)
    best_index = np.argmax(fitness)
    best_solution = population[best_index]
    best_value = np.dot(best_solution, values)
    best_weight = np.dot(best_solution, weights)

    return best_solution, best_value, best_weight, average_fitness_history,
```

Run

In [32]:
```
np.random.seed(42)

best_value, best_weight = -1, -1
best_solution, best_averages, best_bests = None,  None, None

for i in range(100):

  solution, total_value, total_weight, avg_fitness_history, best_fitness_his

  if total_value > best_value and total_weight <= capacity:
    best_value = total_value
    best_weight = total_weight
    best_solution = solution
    best_averages = avg_fitness_history
    best_bests = best_fitness_history


print(f"Solution: {best_solution}")
print(f"Value: {best_value}")
print(f"Weight: {best_weight}")

plot_fitness_over_generations(best_averages, best_bests)
```

```
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7534
Weight: 850
```



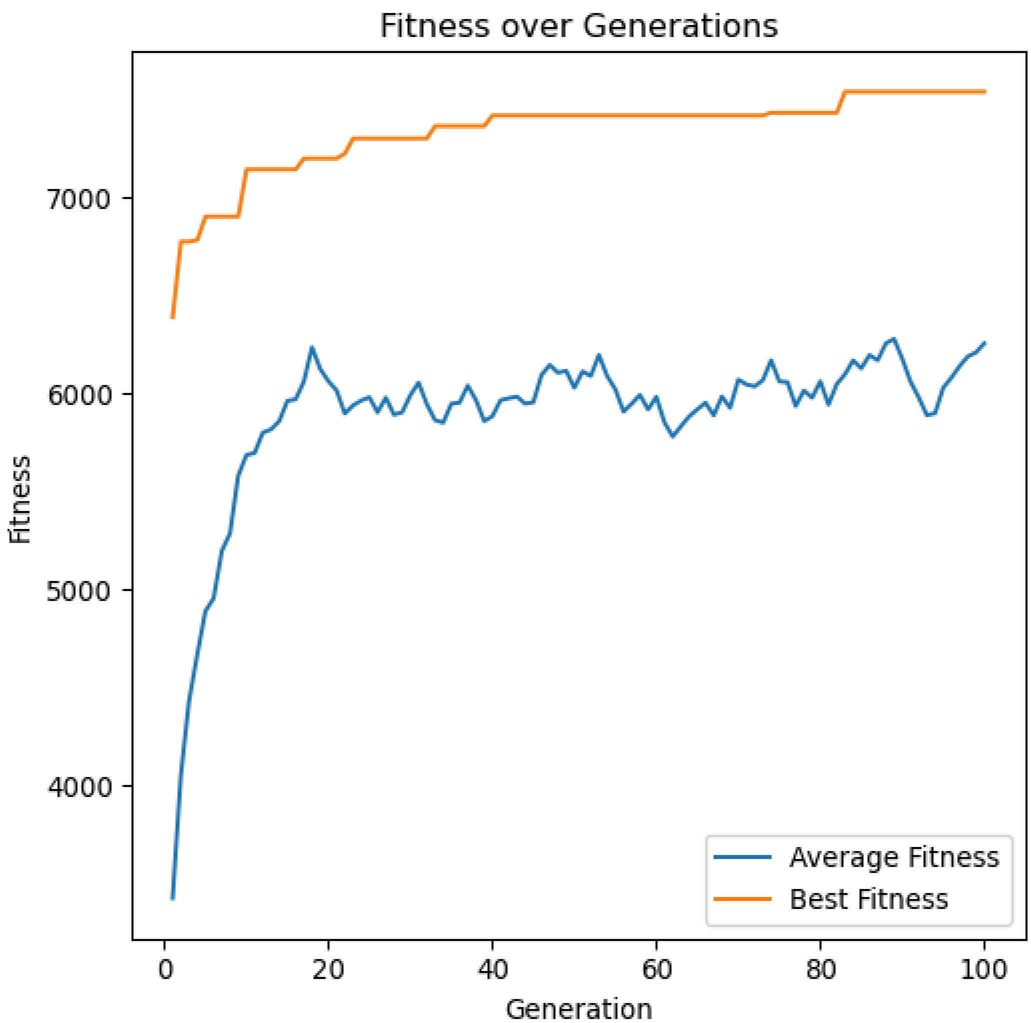
tournament_size=2, 4, 8

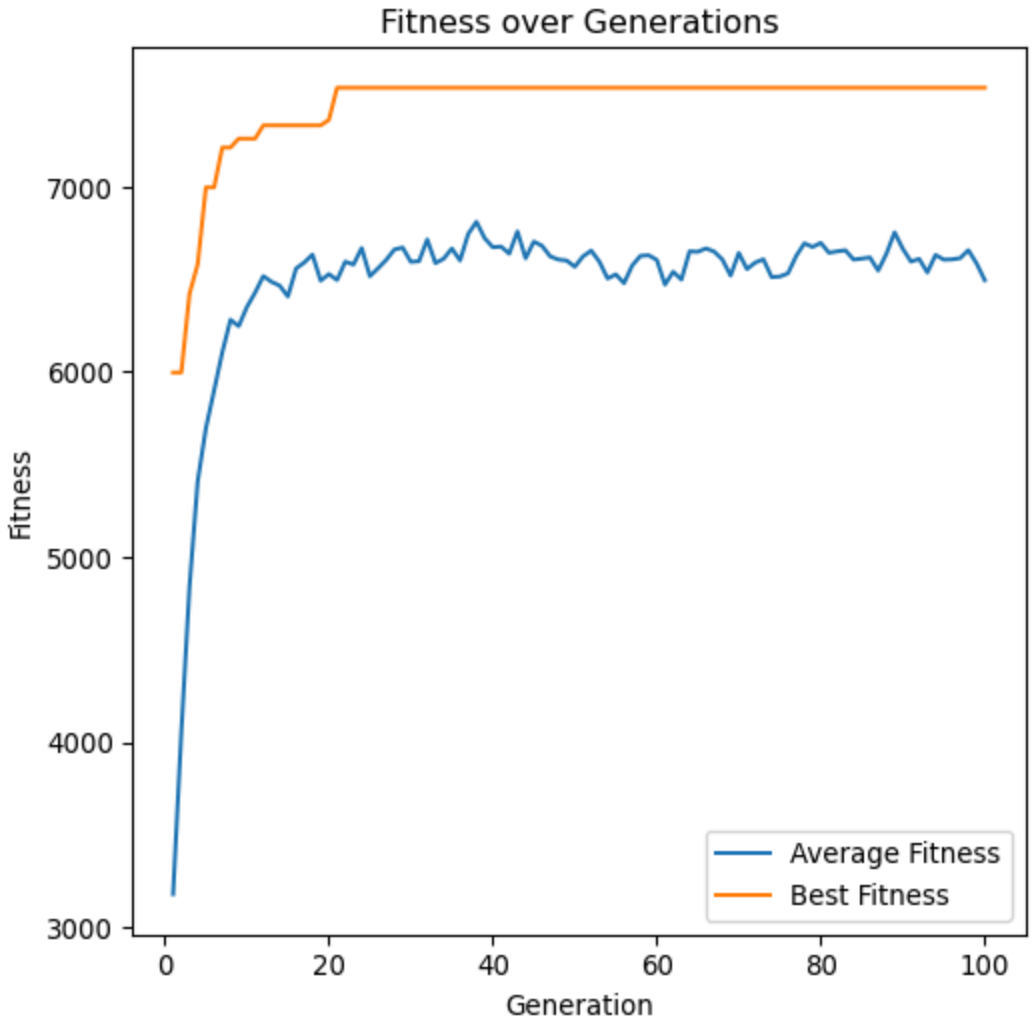
In [33]:

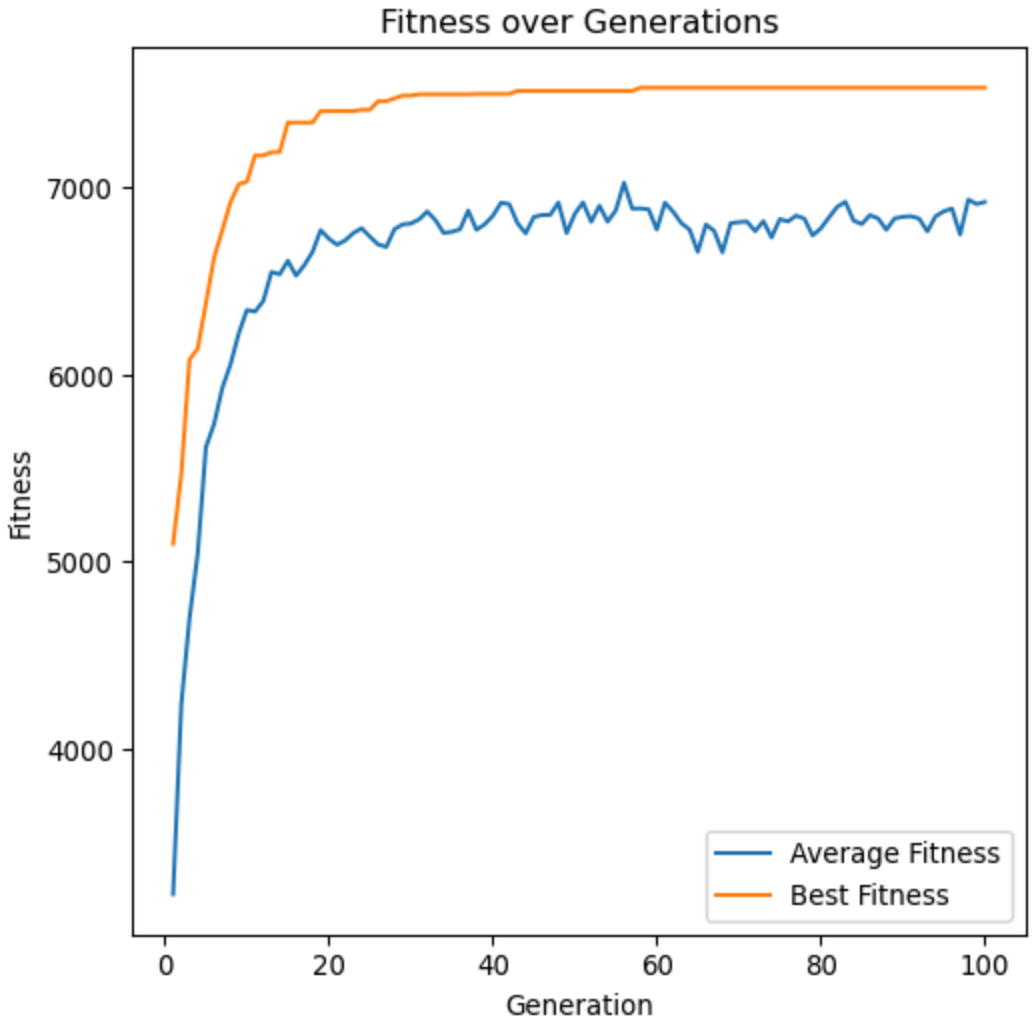
```
np.random.seed(42)

for tournament_size in (2, 4, 8):

  best_value, best_weight = -1, -1
  best_solution, best_averages, best_bests = None,  None, None

  for i in range(100):

      solution, total_value, total_weight, avg_fitness_history, best_fitness
        weights, values, capacity, tournament_size=tournament_size
      )

      if total_value > best_value and total_weight <= capacity:
        best_value = total_value
        best_weight = total_weight
        best_solution = solution
        best_averages = avg_fitness_history
        best_bests = best_fitness_history
```

```python
    print(f"Solution: {best_solution}")
    print(f"Value: {best_value}")
    print(f"Weight: {best_weight}")

    plot_fitness_over_generations(best_averages, best_bests)
```

```
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7534
Weight: 850
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7534
Weight: 850
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7534
Weight: 850
```




Fitness over Generations
Fitness
Generation
7000
6000
5000
4000
3000
0
20
40
60
80
100
Average Fitness
Best Fitness



As one increases the tournament size increases, average and best fitness get closer. Faster convergence.

Crossover

- **Single** or **k-point** cross-over.

K-point crossover is a genetic algorithm technique used for combining two parent solutions to generate offspring. It involves selecting k crossover points and swapping segments of the parents between these points. Here are some scenarios where k-point crossover might be particularly useful:

1. **Complex Solution Spaces**: When the problem domain involves complex interdependencies between variables, k-point crossover can help explore the solution space more thoroughly by allowing multiple segments to be exchanged between parents.

2. **Diverse Genetic Material**: By increasing k, more genetic material from both parents is mixed, which can introduce greater diversity into the offspring. This can be

beneficial in avoiding premature convergence on suboptimal solutions.

3. **Balancing Exploration and Exploitation**: K-point crossover provides a balance between exploration (by introducing new combinations of genetic material) and exploitation (by maintaining some continuity with the parent solutions). This can be advantageous in maintaining a healthy diversity in the population.

4. **Tuning Genetic Algorithm Performance**: Adjusting the number of crossover points (k) allows for fine-tuning the genetic algorithm's performance. A higher k can be used to increase diversity, while a lower k can focus on exploiting known good solutions.

5. **Large Solution Representations**: In problems with large solution representations, such as long binary strings or large arrays, k-point crossover allows for more nuanced mixing of genetic material, potentially leading to more effective search and optimization.

6. **Empirical Testing**: Sometimes, k-point crossover is chosen based on empirical results, where experimentation shows that it performs better for a specific problem or dataset compared to other crossover methods.

K-point crossover is versatile and can be adapted to the needs of different problems by adjusting the number of crossover points, making it a useful tool in the genetic algorithm toolkit.

Uniform crossover

Uniform crossover in genetic algorithms is a recombination technique where **each gene** in the offspring is **independently chosen** from one of the two parent genomes with equal probability. This approach allows for a **more varied combination of parental traits** compared to traditional crossover methods, promoting **greater genetic diversity** in the resulting population.

`uniform_crossover`

In [34]:
```
def uniform_crossover(parents, crossover_rate):

    num_parents, num_genes = parents.shape
    np.random.shuffle(parents)

    offspring = []

    for i in range(0, num_parents, 2):

        parent1 = parents[i]
        parent2 = parents[i+1 if i+1 < num_parents else 0]
```

```
        child1 = parent1.copy()
        child2 = parent2.copy()

        if np.random.rand() < crossover_rate:

            mask = np.random.randint(0, 2, size=num_genes).astype(bool)

            child1[mask], child2[mask] = parent2[mask], parent1[mask]

        offspring.append(child1)
        offspring.append(child2)

    return np.array(offspring)
```

Genetic Algorithm (Version 3)

In [35]:

```
def genetic_algorithm(weights, values, capacity, pop_size=100,
                      num_generations=200, crossover_rate=0.8,
                      mutation_rate=0.05, elite_percent=0.02,
                      selection_type='tournament', tournament_size=3,
                      crossover_type='single_point'):

    num_items = len(weights)
    elite_size = max(1, int(pop_size * elite_percent))
    population = initialize_population(pop_size, num_items)

    average_fitness_history = []
    best_fitness_history = []

    for generation in range(num_generations):

        fitness = evaluate_fitness(population, weights, values, capacity)

        # Track average and best fitness
        average_fitness = np.mean(fitness)
        best_fitness = np.max(fitness)
        average_fitness_history.append(average_fitness)
        best_fitness_history.append(best_fitness)

        # Elitism
        elites = elitism(population, fitness, elite_size)

        # Selection
        if selection_type == 'tournament':
            parents = tournament_selection(population, fitness, tournament_s
        elif selection_type == 'roulette':
            parents = roulette_selection(population, fitness)
        else:
            raise ValueError("Invalid selection type")

        # Crossover
        if crossover_type == 'single_point':
            offspring = single_point_crossover(parents, crossover_rate)
        elif crossover_type == 'uniform':
```

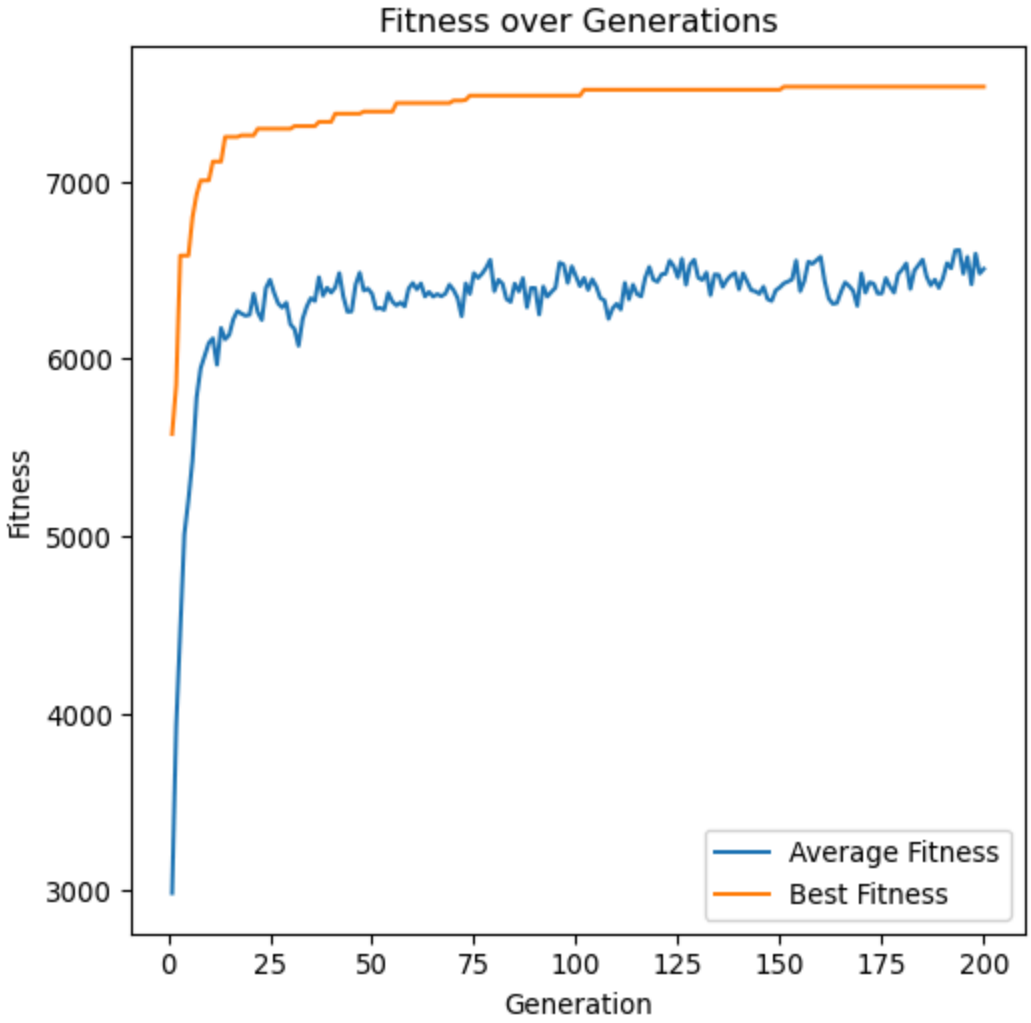
```
            offspring = uniform_crossover(parents, crossover_rate)
        else:
            raise ValueError("Invalid crossover type")

        # Mutation
        offspring = mutation(offspring, mutation_rate)

        # Create new population
        population = np.vstack((elites, offspring))

        # Ensure population size
        if population.shape[0] > pop_size:
            population = population[:pop_size]
        elif population.shape[0] < pop_size:
            # Add random individuals to fill population
            num_new_individuals = pop_size - population.shape[0]
            new_individuals = initialize_population(num_new_individuals, num
            population = np.vstack((population, new_individuals))

    # After all generations, return the best solution
    fitness = evaluate_fitness(population, weights, values, capacity)
    best_index = np.argmax(fitness)
    best_solution = population[best_index]
    best_value = np.dot(best_solution, values)
    best_weight = np.dot(best_solution, weights)

    return best_solution, best_value, best_weight, average_fitness_history,
```

Run

In [36]:

```
np.random.seed(42)

best_value, best_weight = -1, -1
best_solution, best_averages, best_bests = None,  None, None

for i in range(100):

  solution, total_value, total_weight, avg_fitness_history, best_fitness_his
    weights, values, capacity, crossover_type='uniform'
  )

  if total_value > best_value and total_weight <= capacity:
    best_value = total_value
    best_weight = total_weight
    best_solution = solution
    best_averages = avg_fitness_history
    best_bests = best_fitness_history


print(f"Solution: {best_solution}")
print(f"Value: {best_value}")
print(f"Weight: {best_weight}")

plot_fitness_over_generations(best_averages, best_bests)
```

```
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7534
Weight: 850
```



Crossover

Single-Point Crossover is advantageous when **building blocks** (contiguous gene sequences) are **meaningful** and **beneficial to preserve**. However, in the 0/1 Knapsack Problem, the position of items in the chromosome is typically arbitrary, and preserving contiguous sections may not correspond to better solutions.

Uniform Crossover offers **better exploration** by independently mixing genes, which aligns well with the nature of the knapsack problem where each item's inclusion is an independent decision. It reduces positional bias and increases the likelihood of discovering optimal combinations of items.

Crossover

Order preserving. For each offspring, retain the sequence of elements from one parent while filling in the remaining positions with elements from the other parent, preserving their order as they appear in the second parent.

See (Cattolico and Cicirello 2006) for variants.

Order 1 crossover (OX)

Parent 1	1	2	3	4	5	6	7	8	9
Parent 2	9	~~7~~	~~6~~	1	3	5	8	2	4
Selected section copied to child 1									
Child 1				4	5	6	7		
Starts with first gene of parent 2, which is 9. Since 9 does not exist in child 1 yet, it can be copied.									
Child 1	9			4	5	6	7		
The next gene in parent 2 is 7, this already exist in child 1. The next gene is checked, which is 6. It also exists in child 1. The next gene is 1, this is not in child 1 yet and therefore inherited in the next free position.									
Child 1	9	1		4	5	6	7		
The next gene is 3, which can be inherited, and so on. Finally, you get the child below.									
Child 1	9	1	3	4	5	6	7	8	2

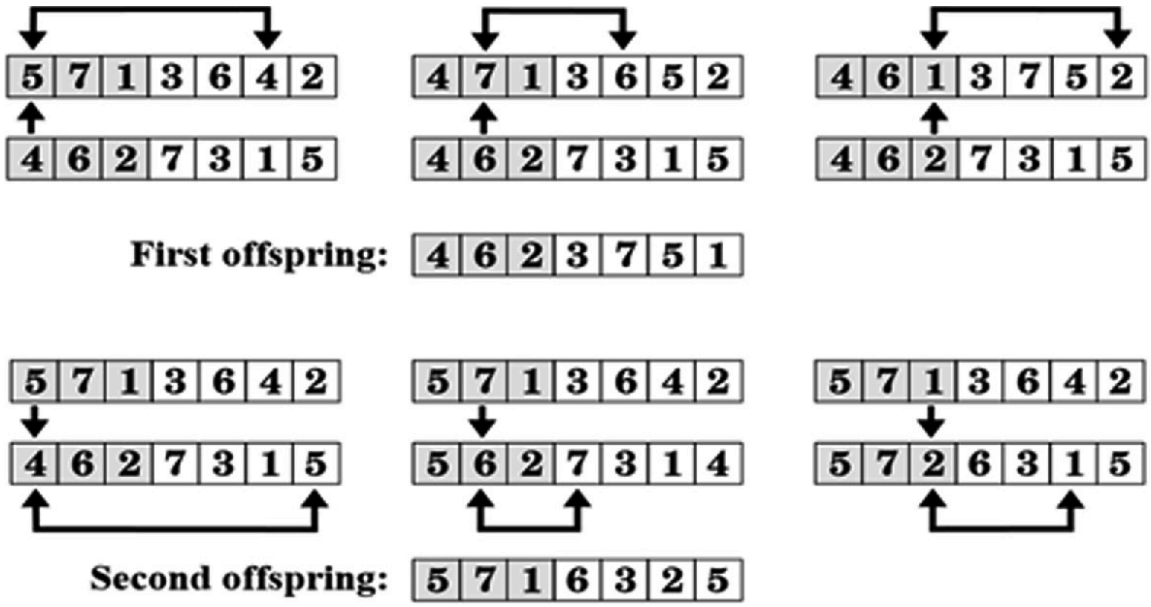
Attribution: (Bye et al. 2021)

Citation: (Davis 2014)

OX is designed to preserve the relative order of elements from one parent while filling gaps with elements from the other parent.

It appears there is an error in the figure: cell 7 of parent 1, located in the first row of the table, should be shaded in gray.

Partially Mapped Crossover (PMX)



Attribution: (Bye et al. 2021)

Citation: (Goldberg and Lingle 2014)

PMX ensures positional consistency by maintaining the relative ordering of elements between parent chromosomes.

Comparison

Feature	PMX	OX
Preservation of Position	High	Low
Preservation of Order	Partial	High
Exploration vs. Exploitation	Exploitation-focused	Exploration-focused
Implementation Complexity	Higher	Lower
Application Suitability	Problems with positional dependencies	Problems with order dependencies

Mutation

- **Bit Flip Mutation:** This involves flipping the value of a bit at a selected position within the chromosome.
    - **Mutation Rate:** A typical mutation rate is $1/n$, where n represents the length of the chromosome.

Mutation

- **Replacement or Random Resetting:** For integer and real-valued chromosomes, employ a random selection from the uniform distribution $U(a, b)$ to choose a new value within the interval $[a, b]$. Similar to the bit flip operator, determine for each position whether a mutation should occur, then apply the replacement if necessary.

Mutation

- **Swap Mutation:** This operator randomly selects two genes within a permutation and exchanges their positions. Such mutations facilitate the exploration of various permutations, potentially yielding improved solutions.

Mutation operators should be crafted to preserve valid permutations while introducing population diversity, thereby mitigating premature convergence to suboptimal solutions.

Remarks

When designing operators, it is critical to ensure they can explore the entire state space exhaustively.

Mutation operators should remain unbiased to maintain the integrity of the exploration process.

Comprehensive Example (1)

A Jupyter Notebook containing all lecture code has been created.

It includes tests on 25 problem instances.

In this context, the **genetic algorithm consistently outperformed the greedy algorithms**, matching the best greedy results in 8 cases and **surpassing them in 17 cases**, with improvements up to 6%.

Comprehensive Example (2)

Fredj Kharroubi conducted an empirical study on the knapsack problem, comparing the performance of several algorithms: **generate-and-test**, **greedy search**, **simulated annealing**, and a **genetic algorithm**.

Jupyter Notebook

Discussions on Hyperparameters

Population Size

- A **larger size** improves genetic diversity but increases computational cost.
- A **smaller size** speeds up execution but risks premature convergence.

Discussions on Hyperparameters

Mutation and Crossover Rates

- **High rates** promote exploration but risk disrupting viable solutions.
- **Low rates** favor exploitation but risk getting stuck in local optima.

Discussions on Hyperparameters

Selection and Tournament Size

- **Tournament selection** is robust against uneven fitness distributions.
- A **medium tournament size (3-5)** balances exploration and exploitation well.

Parameters

Genetic algorithms, like many machine learning and search algorithms, require **hyperparameter tuning** to optimize their performance.

Key hyperparameters in genetic algorithms include **population size**, **mutation rate**, **crossover rate**, **selection method**, and the **number of generations**.

Discussion

Like other metaheuristic approaches, genetic algorithms can become trapped in **local optima**. A common solution, akin to the **random restart** technique used in hill climbing, is to periodically reinitialize the algorithm to explore different regions of the solution space.

Doubling the population size with each restart enhances the likelihood of exploring diverse regions of the state space.

Skepticism toward GA

> ** (Skiena 2008, 267)**

> [I]t is quite **unnatural** to model applications in terms of genetic operators like mutation and crossover on bit strings. The **pseudobiology** adds another level of **complexity** between you and your problem. Second, genetic algorithms **take a very long time on nontrivial problems**. [. . .] [T]he analogy with evolution – where significant progress require [sic] millions of years – can be quite appropriate. [...]
>
> **I have never encountered any problem where genetic algorithms seemed to me the right way to attack it.** Further, I have never seen any computational results reported using genetic algorithms that have favorably impressed me. Stick to simulated annealing for your heuristic search **voodoo** needs.

The **bold formatting** has been applied by me.

It seems that individuals tend to either appreciate or dislike genetic algorithms.

Greedy: value-to-weight ratio

In [37]:
```python
def greedy_knapsack_ratio(weights, values, capacity):

    num_items = len(weights)

    # Calculate value-to-weight ratio for each item
    ratio = values / (weights + 1e-6)

    # Create a list of items with their ratios and original indices
    items = list(zip(ratio, values, weights, range(num_items)))

    # Sort items by ratio in decreasing order
    items.sort(reverse=True)

    total_weight = 0
    total_value = 0
    solution = np.zeros(num_items, dtype=int)

    # Select items based on the sorted order
    for r, v, w, idx in items:
        if total_weight + w <= capacity:
            solution[idx] = 1
            total_weight += w
            total_value += v

    return solution, total_value, total_weight
```

Why adding `1e-6)` to the denominator `ratio = values / (weights + 1e-6)`?

`greedy_knapsack_ratio`

In [38]:
```
solution, total_value, total_weight = greedy_knapsack_ratio(weights, values,

print(f"Solution: {solution}")
print(f"Value: {total_value}")
print(f"Weight: {total_weight}")
```

```
Solution: [1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1
0 1 0 0
 0 1 1 0 1 1 0 1 0 0 1 1 1]
Value: 7534
Weight: 850
```

For **this particular instance**, `greedy_knapsack_ratio` finds the optimal solution!

Frameworks

- DEAP,
  - DEAP is an evolutionary computation framework designed for **rapid prototyping** and testing of ideas. It aims to make algorithms explicit and data structures transparent. The framework seamlessly integrates with **parallelization mechanisms**, including multiprocessing and SCOOP.
  - It has been developed at Université Laval since 2012.
- PyGAD, 5 PyGAD applications

Genetic Programming

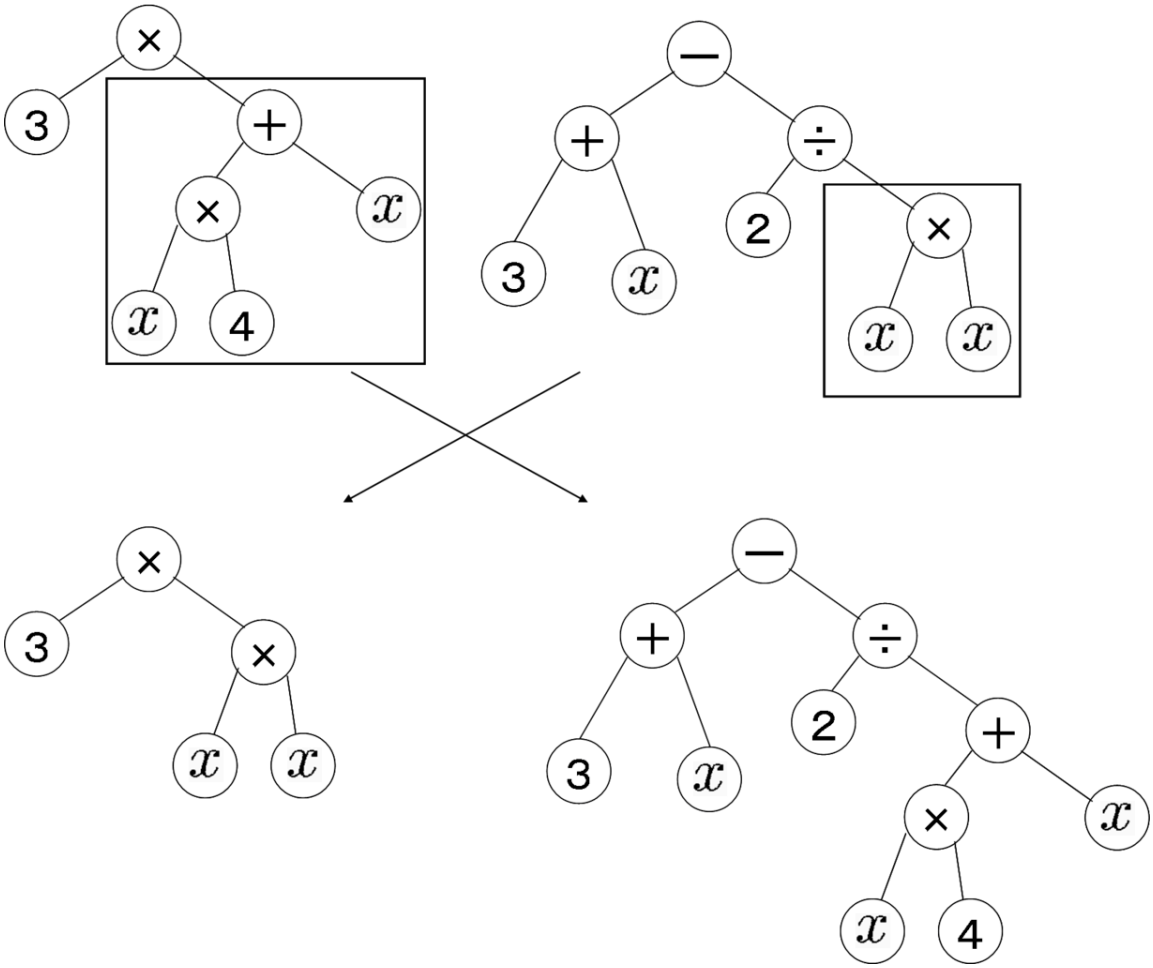
Definition

Genetic programming is an evolutionary algorithm-based methodology that **evolves computer programs** to solve problems by mimicking natural selection processes.

It automatically discovers optimal or near-optimal solutions by iteratively modifying a population of **candidate programs**, guided by a fitness function.

Can be seen as a form of machine learning or automatic programming.

Genetic Programming



Attribution: U-ichi, CC BY-SA 3.0, via Wikimedia Commons

Genetic Programming

Machine Learning Control: Genetic Programming

Prologue

Conclusion

- Rather than exploring a single solution at a time, GA explore several solutions in parallel.

Comparison of SA and GA

Aspect	Simulated Annealing (SA)	Genetic Algorithms (GA)
Solution Representation	Single solution iteratively improved	Population of solutions evolved over generations

Aspect	Simulated Annealing (SA)	Genetic Algorithms (GA)
Exploration Mechanism	Random moves to neighboring solutions; acceptance based on temperature	Crossover and mutation generate new solutions from existing ones
Exploitation Mechanism	Gradual reduction in temperature focuses search around current best solution	Selection and elitism favor fitter individuals in the population
Control Parameters	Temperature, cooling schedule	Population size, crossover rate, mutation rate, selection method
Search Strategy	Explores by accepting worse solutions at higher temperatures	Explores by combining and mutating existing solutions
Balance of Exploration and Exploitation	Controlled by temperature schedule	Controlled by genetic operator rates and selection pressure
Escape from Local Optima	Possible due to probabilistic acceptance of worse solutions	Possible due to diversity in population and genetic variations
Convergence	Depends on cooling schedule; may be slow for large problems	Can converge prematurely without sufficient diversity

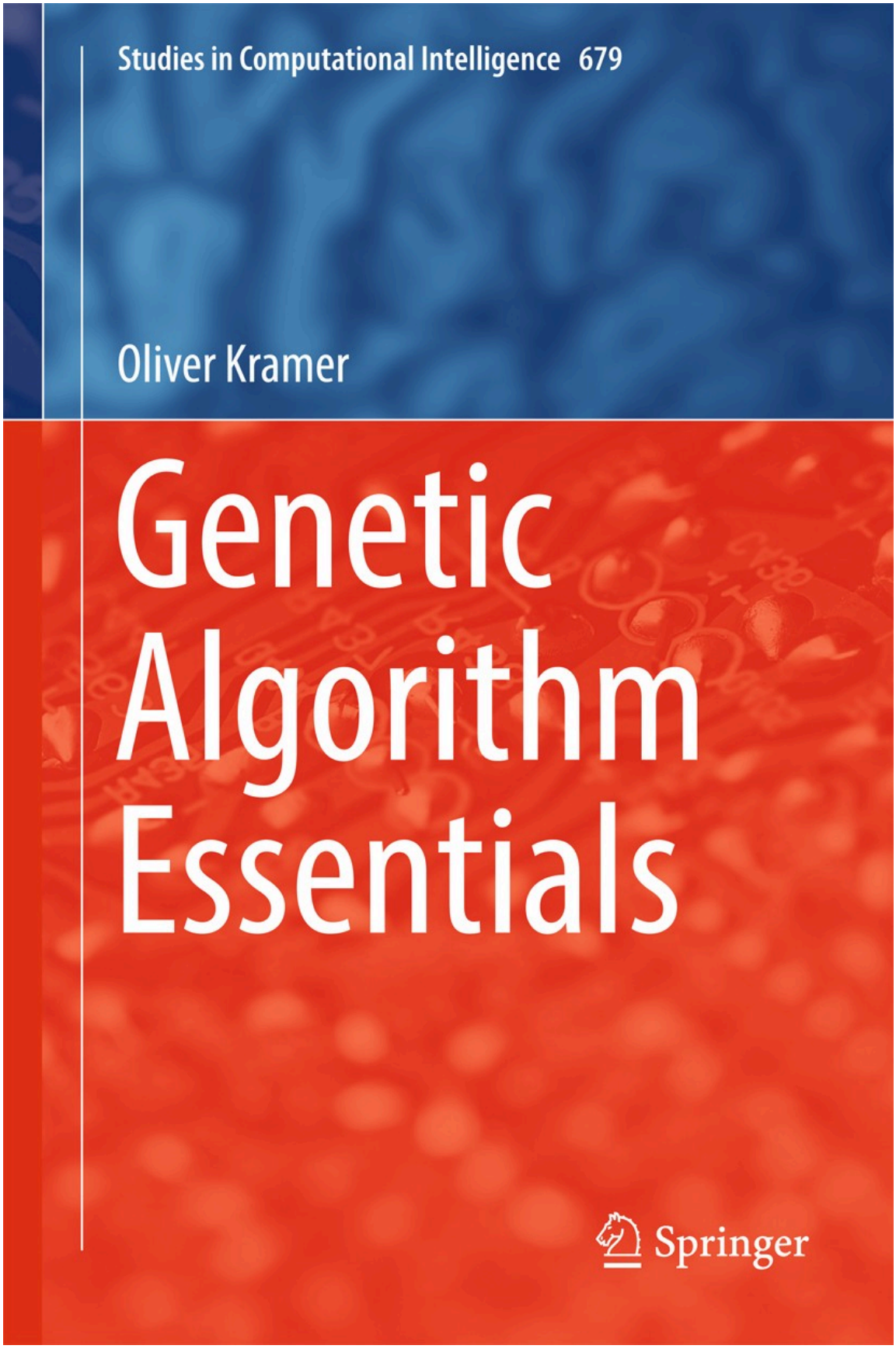
Summary

- **Metaheuristics Overview**
- **Genetic Algorithms (GAs)**
- **Applications of GAs**
- **Components of GAs:**
  - **Encoding:** Representation of candidate solutions (e.g., binary strings for the knapsack problem).
  - **Population:** A set of candidate solutions initialized randomly or by some heuristic.
  - **Selection:** Methods like roulette wheel and tournament selection choose fitter individuals for reproduction.
  - **Crossover:** Combines parts of two parents to create offspring (e.g., single-point crossover).
  - **Mutation:** Randomly alters genes in a chromosome to maintain genetic diversity.
  - **Fitness Function:** Evaluates how close a candidate solution is to the optimum.
- **Knapsack Problem Example:**
  - Demonstrated how to apply GAs to the 0/1 knapsack problem.

    - Provided Python code snippets implementing GA components for the problem.
    - Showed how to generate an initial population, perform crossover and mutation, and select the next generation.
    - Compared GA solutions with optimal solutions obtained using Google's OR-Tools.
- **Metaheuristics Overview:**
    - Metaheuristics are high-level strategies guiding other heuristics to explore large solution spaces.
    - They balance exploitation and exploration to avoid local optima.
    - Genetic algorithms (GAs) are a type of metaheuristic inspired by biological evolution.
- **Genetic Algorithms (GAs):**
    - GAs use a population of candidate solutions (chromosomes) that evolve over generations.
    - Key operations in GAs include selection, crossover (recombination), and mutation.
    - The goal is to optimize a fitness function that measures the quality of solutions.
- **Applications of GAs:**
    - Widely used in optimization, machine learning (e.g., feature selection, hyperparameter tuning), robotics, finance, resource allocation, supply chain, and more.
    - GAs can solve complex problems that are difficult for traditional methods.
- **Components of GAs:**
    - **Encoding:** Representation of candidate solutions (e.g., binary strings for the knapsack problem).
    - **Population:** A set of candidate solutions initialized randomly or by some heuristic.
    - **Selection:** Methods like roulette wheel and tournament selection choose fitter individuals for reproduction.
    - **Crossover:** Combines parts of two parents to create offspring (e.g., single-point crossover).
    - **Mutation:** Randomly alters genes in a chromosome to maintain genetic diversity.
    - **Fitness Function:** Evaluates how close a candidate solution is to the optimum.
- **Knapsack Problem Example:**
    - Demonstrated how to apply GAs to the 0/1 knapsack problem.
    - Provided Python code snippets implementing GA components for the problem.
    - Showed how to generate an initial population, perform crossover and mutation, and select the next generation.

    - Compared GA solutions with optimal solutions obtained using Google's OR-Tools.
- **Choices in GAs:**
    - **Encoding Schemes:** Binary encoding, permutation encoding (for problems like TSP), and value encoding.
    - **Selection Methods:** Roulette wheel selection and tournament selection.
    - **Crossover Techniques:** Single-point, k-point, and order-preserving crossovers.
    - **Mutation Rates:** Balancing exploration and exploitation.
- **Skepticism Toward GAs:**
    - Presented a critical view from Steven Skiena, emphasizing the unnatural modeling and long computation times.
    - Highlighted that some experts prefer other methods like simulated annealing.
- **Genetic Programming:**
    - An extension of GAs where the solutions are computer programs.
    - Programs evolve over time to solve problems, guided by a fitness function.
    - Applications include automated programming and machine learning tasks.
- **Frameworks and Resources:**
    - Mentioned frameworks like DEAP and PyGAD for implementing GAs.
    - Provided resources for further reading and exploration.
    - Encouraged accessing materials through university resources like Springer Link.
- **Conclusion:**
    - GAs explore multiple solutions in parallel, offering a different approach from single-solution methods.
    - Recognized the importance of parameter tuning in GAs for optimal performance.
    - Acknowledged the ongoing debates about the efficacy of GAs in various applications.
- **Next Steps:**
    - Encouraged further exploration of metaheuristics and their applications.
    - Suggested delving into genetic programming for advanced problem-solving techniques.

Further Readings



Kramer (2017), access via Springer Link.

Did you know that you can freely access the entire collection of books from Springer? By using a device connected to a uOttawa IP address and visiting Springer Link, you have the ability to download books in either PDF or EPUB format.

Resources

- Hands on genetic algorithms with Python
- See Direct Evolutionary Optimization of Variational Autoencoders With Binary Latents for a recent application in machine learning.
- AI Improves at Improving Itself Using an Evolutionary Trick: Researchers use evolutionary algorithms to enhance AI coding skills by Matthew Mutson, in IEEE Spectrum, 26 june 2025 (5 minutes read)

Next lecture

- We will look at the Monte Carlo Tree Search (MCTS) algorithm

References

Belmadani, Manuel. 2016. "MotifGP: DNA Motif Discovery Using Multiobjective Evolution." Master of Computer Science, University of Ottawa, School of Electrical Engineering; Computer Science.

Belmadani, Manuel, and Marcel Turcotte. 2016. "MotifGP: Using Multi-Objective Evolutionary Computing for Mining Network Expressions in DNA Sequences." In *CIBCB 2016 - Annual IEEE International Conference on Computational Intelligence in Bioinformatics and Computational Biology*. Chiang Mai, Thailand.

Bye, Robin T., Magnus Gribbestad, Ramesh Chandra, and Ottar L. Osen. 2021. "A Comparison of GA Crossover and Mutation Methods for the Traveling Salesman Problem." In *Innovations in Computational Intelligence and Computer Vision*, edited by Manoj Kumar Sharma, Vijaypal Singh Dhaka, Thinagaran Perumal, Nilanjan Dey, and João Manuel R. S. Tavares, 529–42. Singapore: Springer Singapore.

Cattolico, Mike, and Vincent A Cicirello. 2006. "Non-wrapping order crossover: an order preserving crossover operator that respects absolute position." *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, 1125–32. https://doi.org/10.1145/1143997.1144177.

Davis, Lawrence. 2014. "Job Shop Scheduling with Genetic Algorithms." In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 136–40. Psychology Press.

Dennett, Daniel C. 1995. “Darwin’s Dangerous Idea.” *The Sciences* 35 (3): 34–40.

Eddaly, M., B. Jarboui, and P. Siarry. 2023. *Metaheuristics for Machine Learning: New Advances and Tools*. Computational Intelligence Methods and Applications. Springer Nature Singapore. https://books.google.ca/books?id=yXMtzwEACAAJ.

Georgiev, Bogdan, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. 2025. “Mathematical Exploration and Discovery at Scale.” https://arxiv.org/abs/2511.02864.

Gil-Rios, Miguel-Angel, Ivan Cruz-Aceves, Fernando Cervantes-Sanchez, Igor Guryev, and Juan-Manuel López-Hernández. 2021. “Automatic Enhancement of Coronary Arteries Using Convolutional Gray-Level Templates and Path-Based Metaheuristics.” In *Recent Trends in Computational Intelligence Enabled Research*, edited by Siddhartha Bhattacharyya, Paramartha Dutta, Debabrata Samanta, Anirban Mukherjee, and Indrajit Pan, 129–53. Academic Press.

Goldberg, David E, and Robert Lingle. 2014. “Alleles, Loci, and the Traveling Salesman Problem.” In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 154–59. Psychology Press.

Gregory, T. Ryan. 2009. “Understanding Natural Selection: Essential Concepts and Common Misconceptions.” *Evolution: Education and Outreach* 2 (2): 156–75. https://doi.org/10.1007/s12052-009-0128-1.

Holland, John H. 1973. “Genetic Algorithms and the Optimal Allocation of Trials.” *SIAM Journal on Computing* 2 (2): 88–105. https://doi.org/10.1137/0202009.

———. 1992. “Genetic Algorithms.” *Scientific American* 267: 66–73. https://www.jstor.org/stable/10.2307/24939139.

Kramer, Oliver. 2017. *Genetic Algorithm Essentials*. Studies in Computational Intelligence ; 679. Cham, Switzerland: Springer.

Mayr, Ernst. 1982. *The Growth of Biological Thought: Diversity, Evolution, and Inheritance*. Harvard University Press.

Mitchell, Melanie. 1998. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press.

Oliva, D., E. H. Houssein, and S. Hinojosa. 2021. *Metaheuristics in Machine Learning: Theory and Applications*. Studies in Computational Intelligence. Springer International Publishing. https://books.google.ca/books?id=Zlw4EAAAQBAJ.

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. http://aima.cs.berkeley.edu/.

Santos Amorim, Elisa Portes dos, Carolina Ribeiro Xavier, Ricardo Silva Campos, and Rodrigo Weber dos Santos. 2012. “Comparison Between Genetic Algorithms and Differential Evolution for Solving the History Matching Problem.” In *Computational Science and Its Applications - ICCSA 2012 - 12th International Conference, Salvador de Bahia, Brazil, June 18-21, 2012, Proceedings, Part I*, edited by Beniamino Murgante, Osvaldo Gervasi, Sanjay Misra, Nadia Nedjah, Ana Maria A. C. Rocha, David Taniar, and Bernady O. Apduhan, 7333:635–48. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/978-3-642-31125-3\\48.

Skiena, Steven S. 2008. *The Algorithm Design Manual*. London: Springer. https://doi.org/10.1007/978-1-84800-070-4.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa